TUTORIAL 10

Exploring Arrays, Loops, and Conditional Statements

Creating a Monthly Calendar

OBJECTIVES

Session 10.1
- Create an array
- Work with array properties and methods

Session 10.2
- Create a program loop
- Work with the `for` loop
- Write comparison and logical operators

Session 10.3
- Create a conditional statement
- Use the `if` statement

Case | *The Lyman Hall Theater*

With first-class concerts, performances from Broadway touring companies, and shows from famous comics, singers, and other entertainers, the Lyman Hall Theater is a popular attraction in Brookhaven, Georgia. Lewis Kern is the center's events manager tasked with the job of updating the theater's website.

Lewis wants your help with developing an event calendar application. Rather than constructing the calendar manually, he wants you to write a JavaScript program to automatically generate a web table for a given calendar month, listing the events occurring at the theater during that month. The application should be flexible enough to work with any month so that Lewis only has to enter the event list each month. He wants you to develop a prototype for the August calendar.

STARTING DATA FILES

tutorial
lht_august_txt.html
lht_calendar_txt.js
+ 6 files

review
lht_events_txt.html
lht_table_txt.js
+ 6 files

case1
tc_cart_txt.html
tc_cart_txt.js
tc_order_txt.js
+ 10 files

case2
hg_game_txt.html
hg_report_txt.js
+ 7 files

case3
ah_report_txt.html
ah_report_txt.js
+ 4 files

case4
vw_election_txt.html
vw_results_txt.js
+ 4 files

Session 10.1 Visual Overview:

The createCalendar() function writes the HTML code of the calendar table.

The calCaption() function writes the calendar caption.

An **array** is a collection of values organized under a single name.

```
/* Set the date displayed in the calendar */
var thisDay = new Date("August 24, 2018");

/* Write the calendar to the element with the id 'calendar' */
document.getElementById("calendar").innerHTML = createCalendar(thisDay);

/* Function to generate the calendar table */
function createCalendar(calDate) {
   var calendarHTML = "<table id='calendar_table'>";
   calendarHTML += calCaption(calDate);
   calendarHTML += "</table>";
   return calendarHTML;
}

/* Function to write the calendar caption */
function calCaption(calDate) {
   // monthName array contains the list of month names
   var monthName = ["January", "February", "March", "April",
                    "May", "June", "July", "August", "September",
                    "October", "November", "December"];

   // Determine the current month
   var thisMonth = calDate.getMonth();

   // Determine the current year
   var thisYear = calDate.getFullYear();

   // Write the caption
   return "<caption>" + monthName[thisMonth] + " " + thisYear + "</caption>";

}
```


Values within an array are referenced using the format

array[*i*]

where *array* is the array name and *i* is the index number of the value within the array.

Arrays can be created using the object constructor

var *arrayName* = new Array(*values*);

or using an array literal

var *arrayName* = [*values*];

Creating and Using Arrays

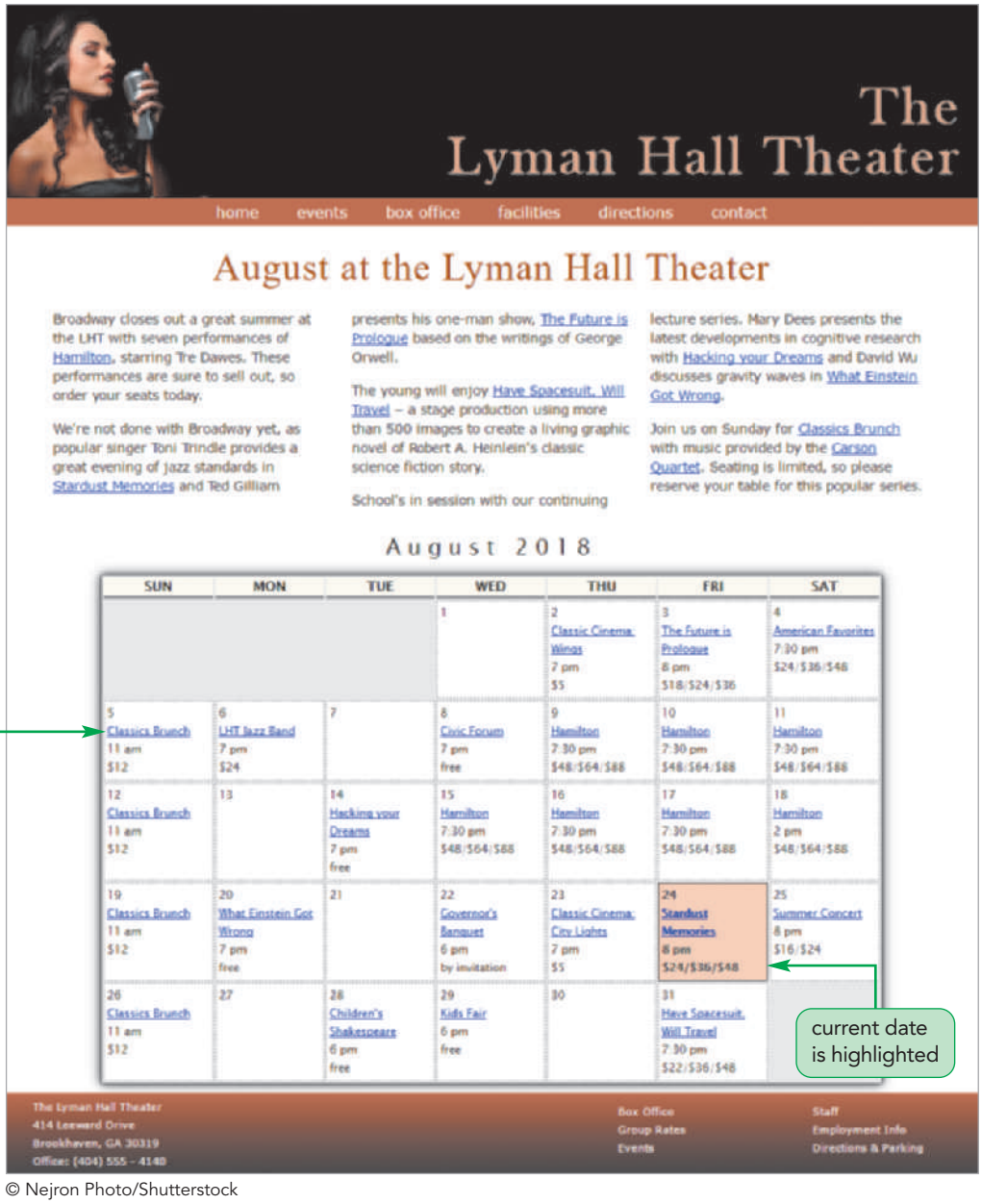
The code written to the web page is:

```
<table id='calendar_table'>
  <caption>August 2018</caption>
</table>
```

Introducing the Monthly Calendar

You and Lewis meet to discuss his idea for a monthly events calendar. He wants the calendar to appear in the form of a web table with links to specific events placed within the table cells. The appearance and placement of the calendar will be set using a CSS style sheet. Figure 10-1 shows a preview of the monthly calendar you will create for the Lyman Hall Theater website.

Figure 10-1 **Monthly events calendar**



© Nejron Photo/Shutterstock

The program you create should be easily adaptable so that it can be used to create other monthly calendars. Lewis wants the code that generates the calendar placed in the lht_calendar.js file. The events listed in the calendar will be placed in the lht_events.js file. Finally, the styles for the calendar will be placed in the lht_calendar.css style sheet file. Lewis already has created the styles required for the calendar table, but he has left the JavaScript coding to you. You will start by adding links to the lht_calendar.js

and lht_calendar.css files to a web page describing the August events at the Lyman Hall Theater. You will work with the lht_events.js file later in this tutorial.

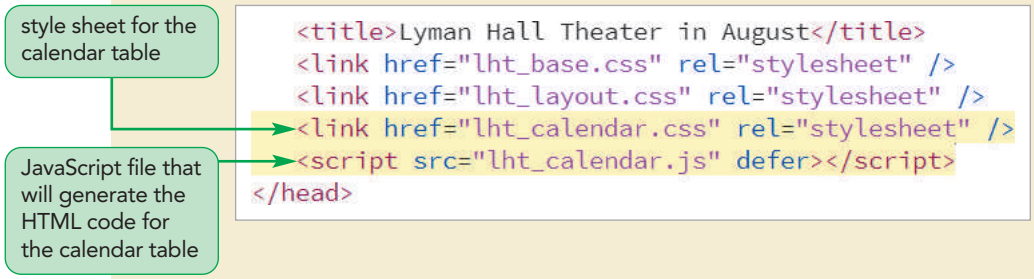
To access the August Events web page:

1. Use your editor to open the **lht_august_txt.html** and **lht_calendar_txt.js** files from the html10 ▸ tutorial folder. Enter ***your name*** and ***the date*** in the comment section of each file and save them as **lht_august.html** and **lht_calendar.js** respectively.
2. Return to the **lht_august.html** file in your editor, and then add the following code above the closing `</head>` tag to create links to both the calendar style sheet and the JavaScript file that will generate the HTML code for the calendar:

```
<link href="lht_calendar.css" rel="stylesheet" />
<script src="lht_calendar.js" defer></script>
```

Figure 10-2 highlights the revised code in the document head.

Figure 10-2 Linking to the style sheet and JavaScript file



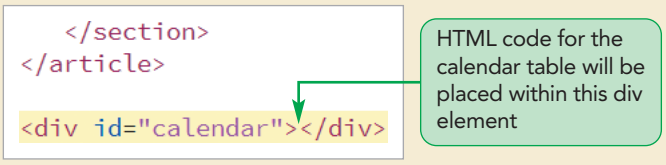
The calendar will be placed within a `div` element with the ID calendar.

3. Scroll down the file and, and, directly below the closing `</article>` tag, insert the following `div` element:

```
<div id="calendar"></div>
```

Figure 10-3 highlights the location where the calendar will be placed.

Figure 10-3 Location of the calendar table



4. Save your changes to the file.

Reviewing the Calendar Structure

The calendar you create will be constructed as a web table. Before you start writing the code to create this table, you should understand the table's structure. Lewis wants the following class names and IDs assigned to the different parts of the table:

- The entire calendar is set in a web table with the ID *calendar_table*.
- The cell containing the calendar title has the ID *calendar_head*.

- The seven cells containing the days of the week abbreviations all belong to the class *calendar_weekdays*.
- The cells containing the dates of the month all belong to the class *calendar_dates*.
- The cell containing the current date has the ID *calendar_today*.

These class and ID designations make it easier for page developers to assign different styles to the different parts of the calendar. If developers want to change the table's appearance, they will not have to edit the JavaScript code to do so; instead, they only will have to modify the style sheet.

Adding the calendar() Function

You will place the commands that generate the calendar within a single function named createCalendar(). The initial code to generate the calendar follows:

```
var thisDay = new Date("August 24, 2018");
document.getElementById("calendar").innerHTML =
createCalendar(thisDay);

function createCalendar(calDate) {
   var calendarHTML = "<table id='calendar_table'>";
   calendarHTML += "</table>";
   return calendarHTML;
}
```

The thisDay variable stores the current date. For the purposes of this example, you will set the date to August 24, 2018. The next line of the function stores the HTML code for the calendar in the `div` element with the ID calendar that you have just created. Initially this HTML code, taken from the createCalendar() function, consists only of the opening and closing tags of the `table` element. Note that you place the value of the `id` attribute within single quotes because the entire text string of HTML code is already enclosed within double quotes.

To insert the initial code of the calendar app:

1. Return to the **lht_calendar.js** file in your editor. Insert following code at the bottom of the file to set the calendar date:

```
/* Set the date displayed in the calendar */
var thisDay = new Date("August 24, 2018");
```

2. Next, add the following code to insert the HTML code of the calendar into the web page:

```
/* Write the calendar to the element with the id "calendar" */
document.getElementById("calendar").innerHTML =
createCalendar(thisDay);
```

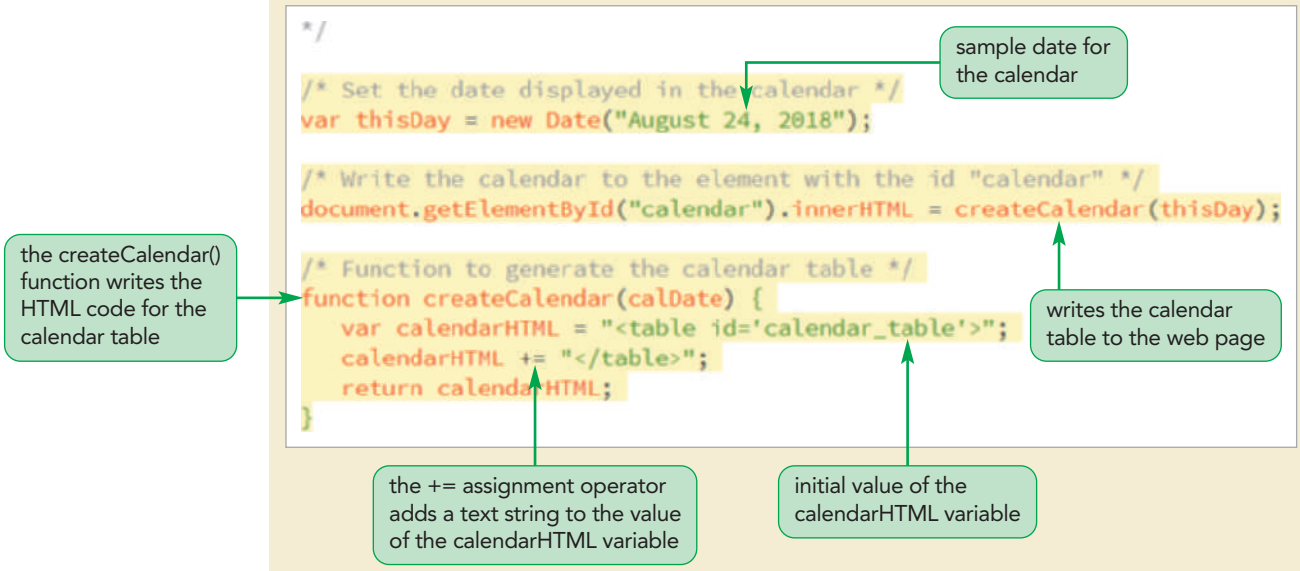
When writing attribute values, you need to enclose the values within single quotes while the text of the HTML code is enclosed within double quotes.

3. Finally, enter the initial code for the createCalendar() function that generates the HTML code:

```
/* Function to generate the calendar table */
function createCalendar(calDate) {
   var calendarHTML = "<table id='calendar_table'>";
   calendarHTML += "</table>";
   return calendarHTML;
}
```

Figure 10-4 describes the code in the file.

Figure 10-4 Initial code for the calendar app



4. Save your changes to the file.

Next, you will start to write the code to create the contents of the calendar table. The three main tasks to complete the calendar table are as follows:

- Create a caption displaying the month and the year
- Create the table row containing the names of the days of the week
- Create rows for each week in the month with cells for each day in the week

In this session, you will learn how to create a calendar table caption. In the next session, you will complete the rest of the table.

Introducing Arrays

Lewis wants the calendar table caption to display the text *Month Year*, where *Month* is the name of the month and *Year* is the four-digit year value. In the last tutorial, you learned that you can use the `getMonth()` method of the JavaScript `Date` object to extract a month number and the `getFullYear()` method to extract the four-digit year value. For example, a `Date` object storing the date March 18, 2018 has a month value of 2 (because month values start with 0 for the month of January) and a four-digit year value of 2018. However, Lewis wants the month name rather than the month number to appear in the table but, because no `Date` method returns the name of the month, you will have to write code to associate each month number with a month name. One way of doing this is by using an array.

An array is a collection of values organized under a single name. Each individual value is associated with a number known as an **index** that distinguishes it from other values in the array. Array values are referenced using the expression

```
array[i]
```

TIP

A common programming mistake with arrays is to use parenthesis symbols ( ) rather than square brackets [ ] to create and reference array values. Remember that only square brackets should be used to reference individual values from an array.

where *array* is the name of the array and *i* is the index of a specific value in the array. Index values start with 0 so that the initial item in an array has an index value of 0, the second item has an index value of 1, and so on. For example, the expression

```
monthName[4]
```

references the fifth (not the fourth) item in the monthName array.

Creating and Populating an Array

To create an array, you can apply the object constructor

```
var array = new Array(length);
```

where *array* is the name of the array and *length* is the number of items in the array. The *length* value is optional; if you omit this parameter, the array expands automatically as more items are added to it. However, by defining the length of an array, JavaScript will allot only the amount of memory needed to generate the array so that the code runs more efficiently. Thus, to create an array named monthName for the 12 month names, you would enter the following statement:

```
var monthName = new Array(12);
```

Alternatively, you could omit the array length and enter the statement as follows:

```
var monthName = new Array();
```

Once you have created an array, you can populate it with values using the same commands you use for any variable. The only difference is that you must specify both the array name and the index number of the array item. The command to set the value of a specific item in an array is

```
array[i] = value;
```

where *value* is the value assigned to the array item with the index value *i*. For example, to insert month names in the monthName array, starting with January, you could enter the following statements:

```
monthName[0] = "January";
monthName[1] = "February";
…
monthName[11] = "December";
```

Rather than writing each array value in a separate statement, you can populate the entire array in a single statement using the following command

```
var array = new Array(values);
```

where *values* is a comma-separated list of the values in the array. The following command places twelve month names into the monthName array in a single statement:

```
var monthName = new Array("January", "February", "March", "April",
"May", "June", "July", "August", "September", "October", "November",
"December");
```

The index numbers are based on the position of the values in the list. The first item in the list (“January”) would have an index number 0, the second (“February”) would have an index of 1, and so forth.

A final way to create an array is with an **array literal**, in which the array values are a comma-separated list within a set of square brackets. The expression to create an array literal is

```
var array = [values];
```

where *values* are the values of the array. The following command uses the array literal form to store an array of month names:

```
var monthName = ["January", "February", "March", "April", "May",
"June", "July", "August", "September", "October", "November",
"December"];
```

TIP

To create an empty array literal that you populate later in the program, leave the brackets blank as in the command var x = [];

If you know the contents of your array, it is usually quicker and easier to set up your array using the array literal notation.

Array values do not need to be the same data type. You can mix numeric values, text strings, and other data types within a single array, as demonstrated by the following statement:

```
var x = ["April", 3.14, true, null];
```

REFERENCE

Creating and Populating Arrays

- To create an array, use the object constructor

  ```
  var array = new Array(length);
  ```

  where *array* is the name of the array and *length* is the number of items in the array. The optional *length* value sets the array to a specified size; if omitted, the array expands as new items are added to it.
- To set the value of an item within an array, use the command

  ```
  array[i] = value;
  ```

  where *i* is the index of the array item and *value* is the value assigned to the item.
- To create and populate an array within a single command, use

  ```
  var array = new Array(values);
  ```

  where *values* is a comma-separated list of values.
- To create an array using the array literal format, use the following statement:

  ```
  var array = [values];
  ```

Now that you have seen how to create and populate an array, you will create an array of month names to use in your calendar application. You will insert the array in a function named calCaption() whose purpose is to write the HTML code of the calendar caption. The function has a single parameter named calDate that stores a `Date` object containing the current date.

To create the calCaption() function:

1. Return to the **lht_calendar.js** file in your editor.
2. At the bottom of the file, insert the following function to write the caption of the calendar table and create the monthName array:

```
/* Function to write the calendar caption */
function calCaption(calDate) {
   // monthName array contains the list of month names
   var monthName = ["January", "February", "March", "April",
                    "May", "June", "July", "August", "September",
                    "October", "November", "December"];
```

3. Next, within the function, use the `getMonth()` and `getFullYear()` methods to extract the month number and 4-digit year number from the calDate parameter by entering the following commands:

```
// Determine the current month
var thisMonth = calDate.getMonth();

// Determine the current year
var thisYear = calDate.getFullYear();
```

4. Finally, complete the function by returning the caption tag for the calendar containing the month name and 4-digit year number. To display the month name, use the monthName array with the value of the thisMonth variable as the index number. Enter the code:

```
    // Write the caption
    return "<caption>" + monthName[thisMonth] + " " + thisYear
+ "</caption>";
}
```

5. Scroll up to the createCalendar() function and insert the following statement directly before the command `calendarHTML += "</table>";`:

```
calendarHTML += calCaption(calDate);
```

This code calls the calCaption() function, which returns the HTML code of the table caption. Figure 10-5 describes the newly added code.

Figure 10-5 The calCaption() function

```
/* Function to generate the calendar table */
function createCalendar(calDate) {
   var calendarHTML = "<table id='calendar_table'>";
   calendarHTML += calCaption(calDate);
   calendarHTML += "</table>";
   return calendarHTML;
}

/* Function to write the calendar caption */
function calCaption(calDate) {
   // monthName array contains the list of month names
   var monthName = ["January", "February", "March", "April",
                    "May", "June", "July", "August", "September",
                    "October", "November", "December"];
   // Determine the current month
   var thisMonth = calDate.getMonth();

   // Determine the current year
   var thisYear = calDate.getFullYear();

   // Write the caption
   return "<caption>" + monthName[thisMonth] + " " + thisYear + "</caption>";
}
```

calls the calCaption() function to insert the HTML code for the table caption

the calCaption() function writes the HTML code for the table caption

creates an array of the month names

extracts the month number of the current month

extracts the 4-digit year value

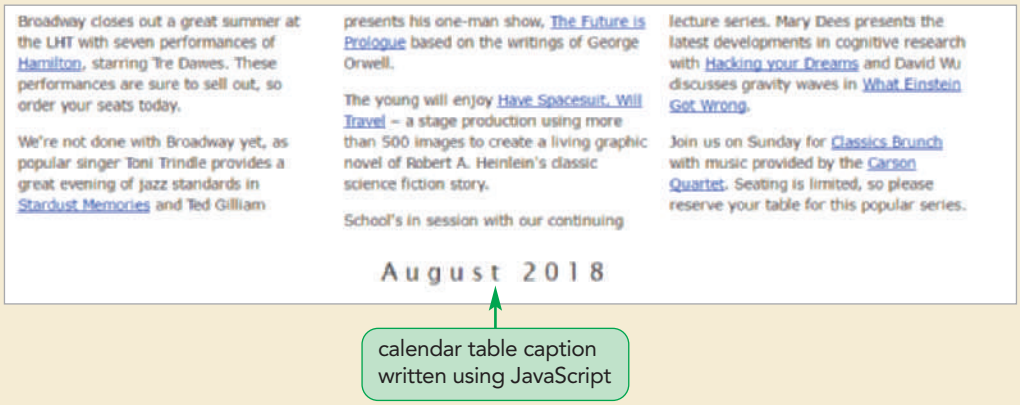
returns the HTML code for the table caption

displays the name of the month drawn from the monthName array

displays the year based on the extracted 4-digit year value

6. Save your changes to the file, and then open **lht_august.html** in your browser. Verify that the web page now shows the caption of the calendar table with the August 2018 date as shown in Figure 10-6.

Figure 10-6 Calendar caption displayed in the web page



Trouble? If the caption does not appear in the page, your code might contain a mistake. Check your code against the code shown in the previous figures. Common sources of error include forgetting to close all quoted text strings, failing to match the use of uppercase and lowercase letters in function names and variable names, misspelling function names and variable names, and failing to close parentheses and brackets when required.

Next, you will explore the properties and methods associated with arrays.

Working with Array Length

A JavaScript array automatically expands in length as more items are added. To determine the array's current size, apply the following `length` property

```
array.length
```

where *array* is the name of the array. The value returned by the `length` property is equal to one more than the highest index number in the array (because array indices start at 0 rather than 1), so, if the highest number in the index is 11, then the value returned would be 12.

JavaScript allows for the creation of **sparse arrays**, in which some array values are undefined. As a result, the `length` value is not always the same as the number of array values. For example, the following commands create a sparse array in which only the first and last items have defined values:

```
var x = new Array();
x[0] = "Lewis";
x[99] = "80517";
```

TIP

You can add new items to the end of any array using the command *array*[*array.length*] = *value*;

The value of the `length` property for this array is 100 even though it only contains two values. Sparse arrays occur frequently in database applications involving customer records where items such as mobile phone numbers or postal codes have not been entered for every person.

REFERENCE

Specifying Array Length

- To determine the size of an array, use the property

  ```
  array.length
  ```

  where *array* is the name of the array and `length` is one more than the highest index number in the array.
- To add an item to the end of an array, run the command

  ```
  array[i] = value;
  ```

  where *i* is an index value higher than the highest index currently in the array. If you don't know the highest index number, use the property *array*.`length` in place of *i*.
- To remove items from an array, run the command

  ```
  array.length = value;
  ```

  where *value* is an integer that is smaller than the highest index currently in the array.

Note that you cannot reduce the value of the `length` property without removing items from the end of your array. For example, the following command would reduce the monthName array to the first three months—January, February, and March:

```
monthName.length = 3;
```

Increasing the value of the `length` property adds more items to an array, but the items have null values until they are defined.

PROSKILLS

Problem Solving: Using Multidimensional Arrays

Many database applications need to store data in a rectangular format known as a **matrix**, in which the values are arranged in a rectangular grid. The following is an example of a matrix laid out in a grid of three rows and four columns:

$$\begin{bmatrix} 4 & 15 & 8 & 2 \\ 1 & 3 & 18 & 6 \\ 3 & 7 & 10 & 4 \end{bmatrix}$$

The rows and columns in a matrix form the basis for indices. For example, the value 18 from this matrix is referenced using the index pair (2, 3) because the value 18 appears at the intersection of the second row and third column.

Although matrices are commonly used in databases (where each row might represent an individual and each column a characteristic of that individual), JavaScript does not support matrices. However, you can mimic the behavior of matrices in JavaScript by nesting one array within another in a structure called a **multidimensional array**. For example, the following code creates the array *mArray*, which contains a collection of nested arrays:

```
var mArray =
   [
     [4, 15, 8, 2],
     [1, 3, 18, 6],
     [3, 7, 10, 4]
   ];
```

Note that the values of this array match the values of the matrix shown above. In this case, the first nested array matches the first row of the matrix, the second array matches the second row, and the third array matches the third row. The values of the nested arrays are matched with each of the four columns.

Values within a multidimensional array are referenced by the expression

```
array[x][y]
```

where ***x*** contains the index of the outer array (the row) and ***y*** contains the index of the nested array (the column). Thus, the expression

```
mArray[1][2]
```

returns the value 18 from the matrix's second row and third column (remember that indices start with 0, not 1). The number of rows in a multidimensional array is given by the `length` property. The number of columns can be determined by retrieving the `length` property for the first row of the table. For example, the expression

```
mArray[1].length
```

would return a value of 4 for the fours columns in mArray. Note that this approach presumes that every row has the same number of columns. You can continue to nest arrays in this fashion to create matrices of even higher dimensions.

Reversing an Array

Arrays are associated with a collection of methods that allow you to change their content, order, and size. You can also use these methods to combine different arrays into a single array and to convert arrays into text strings. Although you will not need to use these methods in the calendar app, you will examine them for future projects.

By default, items are placed in an array either in the order in which they are defined or explicitly by index number. JavaScript supports two methods for changing the order of these items: `reverse()` and `sort()`. The `reverse()` method, as the name suggests, reverses the order of items in an array, making the last items first and the first items last. In the following set of commands, the `reverse()` method is used to change the order of the values in the weekDay array:

```
var weekDay = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
weekDay.reverse();
```

After running the `reverse()` method, the weekDay array would contain the items in the following order: "Sat", "Fri", "Thu", "Wed", "Tue", "Mon", and finally, "Sun".

Sorting an Array

The `sort()` method rearranges array items in alphabetical order. This can cause unexpected results if you apply the `sort()` method to data values that are not usually sorted alphabetically. Applying the `sort()` method to numeric values, will sort the values in order by their leading digits, rather than by their numerical values. Thus, applying the `sort()` method in the following set of commands

```
var x = [3, 45, 1234, 24];
x.sort();
```

would result in the order 1234, 24, 3, 45 because this is the order of those numbers when sorted by their leading digits. To correctly sort numeric data, you must create a **compare function** that compares the values of two adjacent array items. The general form of a compare function is

```
function fname(a, b) {
   return a negative, positive, or 0 value
}
```

where *fname* is the name of the compare function, and *a* and *b* are parameters that represent a pair of array values. The function then returns a negative, positive, or zero value based on the comparison of those values. If a negative value is returned, then *a* is placed before *b* in the array. If a positive value is returned, then *b* is placed before *a*, and finally, if a zero value is returned, *a* and *b* retain their original positions. The compare function is applied to every pair of values in the array to ensure they are sorted in the proper order.

The following compare function could be used to sort numeric values in ascending order

```
function ascending(a, b) {
   return a - b;
}
```

whereas to sort numbers in a descending order, you could apply the following function, which subtracts *a* from *b*, rather than *b* from *a*:

```
function descending(a, b) {
   return b - a;
}
```

Other compare functions are possible to deal with a wide variety of sorting rules, but these two are the simplest for sorting arrays of numeric values.

The compare function is applied to the `sort()` method as follows

```
array.sort(fname)
```

TIP

You can also sort an array in descending order by sorting it first in ascending order and then by applying the `reverse()` method to reverse the sorted order of the array.

where *fname* is the name of the compare function. For example, to use the ascending() compare function to sort the x array described earlier in ascending numeric order, you would run the following command:

```
x.sort(ascending)
```

After applying the `sort()` method with the ascending function, the values in the x array would be sorted in ascending numeric order as: 3, 24, 45, and finally, 1234.

INSIGHT

Performing a Random Shuffle

For some applications, you will want to randomly rearrange the contents of an array. For example, you might be writing a program to simulate a randomly shuffled deck of cards. You can shuffle an array using the same `sort()` method you use to place the array in a defined order; however, to place the items in a random order, you use a compare function that randomly returns a positive, negative, or 0 value. The following compare function employs a simple approach to this problem:

```
function randOrder(){
  return 0.5 - Math.random();
}
```

The following code demonstrates how this compare function could be used to randomly shuffle an array of poker cards:

```
var pokerDeck = new Array(52);
pokerDeck[0] = "2 of Clubs";
pokerDeck[1] = "3 of Clubs";
…
pokerDeck[51] = "Ace of Spades";
pokerDeck.sort(randOrder)
```

After running this command, the contents of the pokerDeck array will be placed in random order. To reshuffle the array, you would simply rerun the `sort()` method with the randOrder() function.

Extracting and Inserting Array Items

In some scripts, you might want to extract a section of an array, known as a **subarray**. One way to create a subarray is with the following `slice()` method

```
array.slice(start, stop)
```

where *start* is the index value of the array item at which the slicing starts and *stop* is the index value at which the slicing ends. Note that the stop index value is not included in the subarray. The *stop* value is optional; if it is omitted, the array is sliced to its end. The original contents of the array are unaffected after slicing, but the extracted items can be stored in another array. For example, the following command slices the monthName array, extracting only three summer months—June, July, August—and storing them in the summerMonths array:

```
summerMonths = monthName.slice(5, 8);
```

Remember that arrays start with the index value 0, so the sixth month of the year (June) has an index value of 5 and the ninth month of the year (September) has an index value of 8.

Related to the `slice()` method is the following `splice()` method

```
array.splice(start, size, values)
```

which is a general-purpose method for removing and inserting array items, where *start* is the starting index in the array, *size* is the number of array items to remove after the *start* index, and *values* is an optional comma-separated list of values to insert into the array. If no *values* are specified, the splice method simply removes items from the array.

The following statement employs the splice() method to remove the summer months from the monthName array:

```
summerMonths = monthName.splice(5, 3);
```

However, to insert new abbreviations of month names into the monthName array, you could apply the following splice() method which places the values "Jun", "Jul", and "Aug" into the array starting with the 5th index number:

```
monthName.splice(5, 3, "Jun", "Jul", "Aug");
```

The important difference between the slice() and splice() methods is that the splice() method always alters the original array, so you should not use the splice() method if you want the original array left unaffected.

Using Arrays as Data Stacks

Arrays can be used to store information in a data structure known as a **stack** in which new items are added to the top of the stack—or to the end of the array—much like a person clearing a dinner table adds dishes to the top of a stack of dirty plates. A stack data structure employs the **last-in first-out (LIFO)** principle in which the last items added to the stack are the first ones removed. You encounter stack data structures when using the Undo feature of some software applications, in which the last command you performed is the first command that is undone.

JavaScript supports several methods to allow you to work with a stack of array items. For example, the push() method appends new items to the end of an array. It has the syntax

```
array.push(values)
```

where *values* is a comma-separated list of values to be appended to the end of the array. To remove—or **unstack**—the last item, you apply the pop() method, as follows:

```
array.pop()
```

The following set of commands demonstrates how to use the push() and pop() methods to employ the LIFO principle by adding and then removing items from a data stack:

```
var x = ["a", "b", "c"];
x.push("d", "e"); // x = ["a", "b", "c", "d", "e"]
x.pop();          // x = ["a", "b", "c", "d"]
x.pop();          // x = ["a", "b", "c"]
```

In this code, the push() method adds two items to the end of the array, and then the pop() method removes those last items one at a time.

A **queue**, which employs the **first-in-first-out (FIFO)** principle in which the first item added to the data list is the first removed, is similar to a stack. You see the FIFO principle in action in a line of people waiting to be served. For array data that should be treated as a queue, you use the shift() method, which is similar to the pop() method except that it removes the first array item, not the last item. JavaScript also supports the unshift() method, which inserts new items at the front of the array.

REFERENCE

Using Array Methods

- To reverse the order of items in an array, use the method

  ```
  array.reverse()
  ```

  where *array* is the name of the array.
- To sort an array in alphabetical order, use the following method:

  ```
  array.sort();
  ```

- To sort an array in any order, use

  ```
  array.sort(fname)
  ```

  where *fname* is the name of a compare function that returns a positive, negative, or 0 value.
- To extract items from an array without affecting the array contents, use

  ```
  array.slice(start, stop)
  ```

  where *start* is the index of the array item at which the slicing starts and *stop* is the index at which the slicing ends. If no *stop* value is provided, the array is sliced to the end of the array.
- To remove items from an array, use

  ```
  array.splice(start, size)
  ```

  where *start* is the index of the array item at which the splicing starts and *size* is the number of items to remove from the array. If no *size* value is specified, the array is spliced to its end.
- To replace items in an array, use

  ```
  array.splice(start, size, values)
  ```

  where *values* is a comma-separated list of new values to replace the old values in the array.
- To add new items to the end of an array, use

  ```
  array.push(values)
  ```

  where *values* is a comma-separated list of values.
- To remove the last item from an array, use the following method:

  ```
  array.pop()
  ```

Figure 10-7 summarizes several other methods that can be applied to arrays. Arrays are a powerful and useful feature of the JavaScript language. The methods associated with arrays can be used to simplify and expand the capabilities of web page scripts.

Figure 10-7 Array methods

Method	Description
`copyWithin(target, start[, end])`	Copies items within the array to the *target* index, starting with the *start* index and ending with the optional *end* index
`concat(array1, array2,...)`	Joins the array to two or more arrays, creating a single array containing the items from all the arrays
`fill(value[, start][, end])`	Fills the array with items having the value *value*, starting from the *start* index and ending at the *end* index
`indexOf(value[, start])`	Searches the array, returning the index number of the first element equal to *value*, starting from the optional *start* index
`join(separator)`	Joins all items in the array into a single text string; the array items are separated using the text in the *separator* parameter; if no *separator* is specified, a comma is used
`lastIndexOf(value[, start])`	Searches backward through the array, returning the index number of the first element equal to *value*, starting from the optional *start* index
`pop()`	Removes the last item from the array
`push(values)`	Appends the array with new items, where *values* is a comma-separated list of item values
`reverse()`	Reverses the order of items in the array
`shift()`	Removes the first item from the array
`slice(start, stop)`	Extracts the array items starting with the *start* index up to the *stop* index, returning a new subarray
array`.splice(start, size, values)`	Extracts *size* items from the array starting with the item with the index *start*; to insert new items into the array, specify the array items in a comma-separated *values* list
array`.sort(fname)`	Sorts the array where *fname* is the name of a function that returns a positive, negative, or 0 value; if no `function` is specified, *array* is sorted in alphabetical order
array`.toString()`	Converts the contents of the array to a text string with the array values in a comma-separated list
array`.unshift(values)`	Inserts new items at the start of the array, where *values* is a comma-separated list of new values

You set up the first parts of the online calendar in this session. In the next sessions, you will complete the monthly calendar by working with loops and conditional statements.

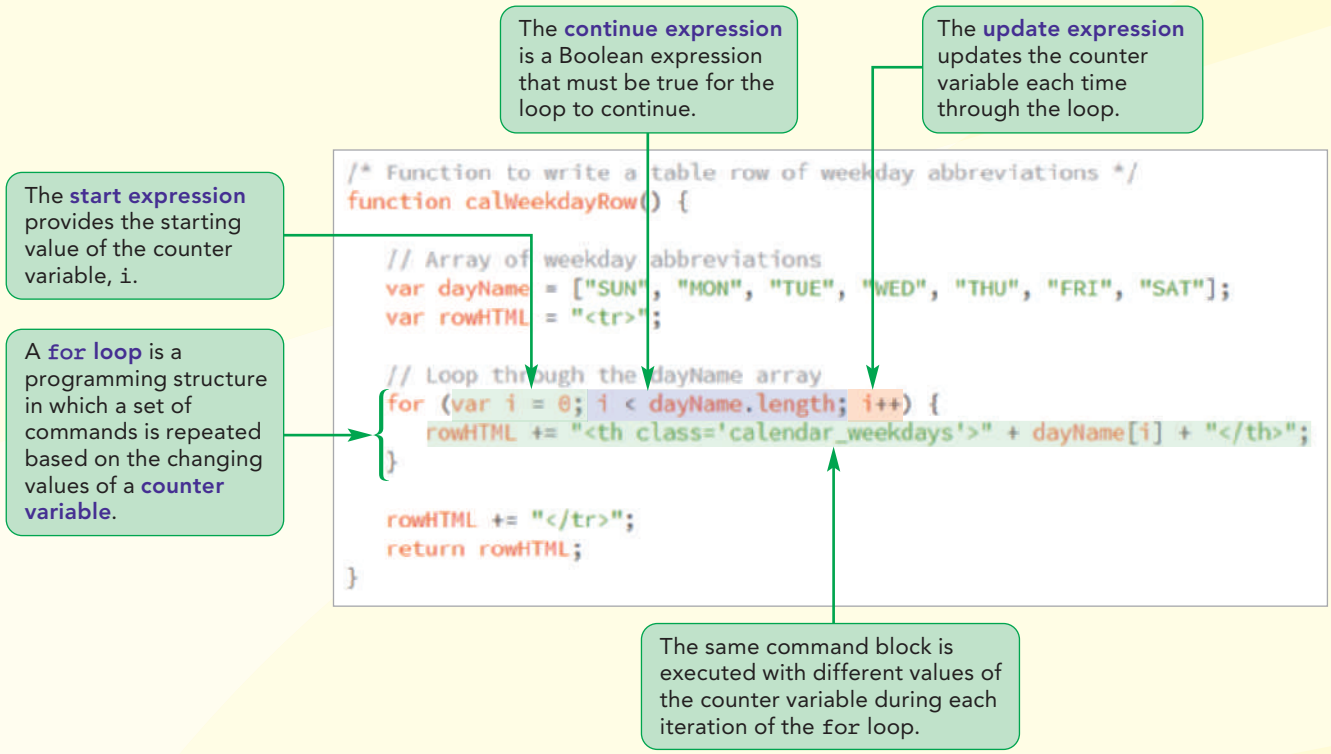
REVIEW

Session 10.1 Quick Check

1. What is an array?
2. Provide a command to create an array named dayNames using the object constructor form.
3. Provide a command to create and populate the dayNames array with the abbreviations of the seven days of the week (starting with Sun and going through Sat). Use the array literal form.
4. Provide a command to return the third value from the dayNames array.
5. Provide a command to sort the dayNames array in alphabetical order.
6. Provide a command to extract the middle five values from the dayNames array.
7. Provide a command to create a multidimensional array named myArray for the following matrix:

$$\begin{bmatrix} 3 & 0 & -2 \\ 12 & -8 & 1 \\ 4 & 1 & -3 \end{bmatrix}$$

Session 10.2 Visual Overview:



Applying a Program Loop

The days of the week are written using a **program loop** that repeats a set of similar commands until a stopping condition is met.

The code created for the table row is as follows:

```
<tr>
   <th class='calendar_weekdays'>SUN</th>
   <th class='calendar_weekdays'>MON</th>
   <th class='calendar_weekdays'>TUE</th>
   <th class='calendar_weekdays'>WED</th>
   <th class='calendar_weekdays'>THU</th>
   <th class='calendar_weekdays'>FRI</th>
   <th class='calendar_weekdays'>SAT</th>
</tr>
```

Working with Program Loops

Now that you are familiar with the properties and methods of arrays, you will return to working on the calendar app. So far, you have created only the table caption displaying the calendar's month and year. The first row of the table will contain the three-letter abbreviations of the seven days of the week, starting with SUN and continuing through SAT. Each abbreviation needs to be placed within an element with the class name calendar_weekdays using the following code:

```
<tr>
   <th class='calendar_weekdays'>SUN</th>
   <th class='calendar_weekdays'>MON</th>
   <th class='calendar_weekdays'>TUE</th>
   <th class='calendar_weekdays'>WED</th>
   <th class='calendar_weekdays'>THU</th>
   <th class='calendar_weekdays'>FRI</th>
   <th class='calendar_weekdays'>SAT</th>
</tr>
```

This code contains a lot of repetitive text with the same `th` element and class name repeated seven times. Imagine if you had to repeat essentially the same string of code dozens, hundreds, or even thousands of times—the code would become unmanageably long. Programmers deal with this kind of situation by creating program loops. A **program loop** is a set of commands executed repeatedly until a stopping condition is met. Two commonly used program loops in JavaScript are `for` loops and `while` loops.

Exploring the `for` Loop

In a `for` loop, a variable known as a counter variable is used to track the number of times a block of commands is run. Each time through the loop, the value of the counter variable is increased or decreased by a set amount. When the counter variable reaches or exceeds a specified value, the `for` loop stops. The general structure of a `for` loop is

```
for (start; continue; update) {
   commands
}
```

where *`start`* is an expression that sets the initial value of a counter variable, *`continue`* is a Boolean expression that must be true for the loop to continue, *`update`* is an expression that indicates how the value of the counter variable should change each time through the loop, and *`commands`* are the JavaScript statements that are run for each loop.

Suppose you want to set a counter variable to range in value from 1 to 4 in increments of 1. You could use the following expression to set the initial value of the counter variable:

```
var i = 1;
```

The name of the counter variable in this example is `i`, which is a common variable name often applied in program loops.

The next expression in the `for` loop structure defines the condition under which the program loop continues. The following expression sets the loop to continue as long as the value of the counter variable is less than or equal to 4:

```
i <= 4;
```

Finally, the following update expression uses the increment operator to indicate that the value of the counter variable increases by 1 each time through the program loop:

```
i++;
```

Putting all of these expressions together, you get the following `for` loop:

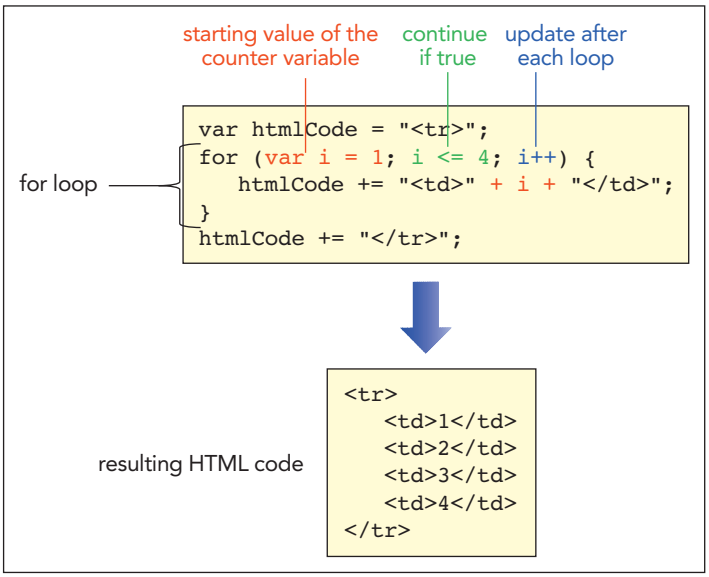
```
for (var i = 1; i <= 4; i++) {
   commands
}
```

The collection of commands that is run each time through a loop is known collectively as a **command block**, a feature you have already worked with in functions. A command block is indicated by its opening and closing curly braces `{ }`. The following is an example of a `for` loop that adds the HTML code for four `td` elements to a table row:

```
var htmlCode = "<tr>";
for (var i = 1; i <= 4; i++) {
   htmlCode += "<td>" + i + "</td>";
}
htmlCode += "</tr>";
```

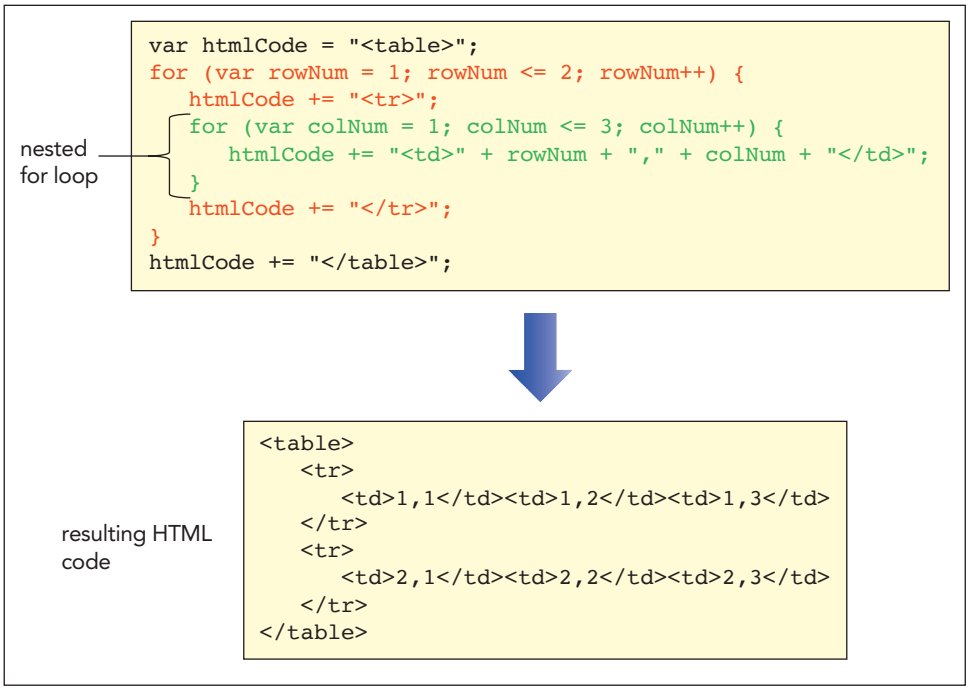
As shown in Figure 10-8, each time through the loop, the value displayed in the table cell is changed by 1.

Figure 10-8 Writing HTML code with a for loop



One `for` loop can be nested within another. Figure 10-9 shows the code used to create a table with two rows and three columns.

Figure 10-9 Nested for loop



This example uses two counter variables named rowNum and colNum. The rowNum variable loops through the values 1 and 2 and for each of those values, the colNum variable loops through the values 1, 2, and 3. Each time the value of the colNum variable changes, a new cell is added to the table. Each time the value of the rowNum variable changes, a new row is added to the table.

The update expression is not limited to increasing the counter by 1. You can use the other operators introduced in the previous tutorial to create a wide variety of increment patterns. Figure 10-10 shows a few of the many different ways of updating the value of the counter variable in a `for` loop.

Figure 10-10 for loop counter values

for Loop	Counter Values
`for (var i = 1; i <= 5; i++)`	i = 1, 2, 3, 4, 5
`for (var i = 5; i > 0; i--)`	i = 5, 4, 3, 2, 1
`for (var i = 0; i <= 360; i+=60)`	i = 0, 60, 120, 180, 240, 360
`for (var i = 1; i <= 64; i*=2)`	i = 1, 2, 4, 8, 16, 32, 64

Exploring the `while` Loop

The `for` loop is only one way of creating a program loop in JavaScript. The **`while` loop**, in which a command block is run as long as a specific condition is met, is similar to the `for` loop. However, unlike the `for` loop, the condition in a `while` loop does not depend on the value of a counter variable. The `while` loop has the general syntax

```
while (continue) {
   commands
}
```

where *`continue`* is a Boolean expression that must be true for the command block to be run; otherwise, the command block is skipped and the program loop ends.

The following code shows how to create the table shown earlier in Figure 10-8 as a `while` loop:

```
var htmlCode = "<tr>";
var i = 1;
while (i <= 4) {
   htmlCode += "<td>" + i + "</td>";
   i++;
}
```

The `while` loop continues as long as the value of the `i` variable remains less than or equal to 4. Each time through the command block, the loop writes the value of `i` into a table cell and then increases the counter by 1.

Like `for` loops, `while` loops can be nested within one another. The following code demonstrates how to create the 2 × 3 table shown earlier in Figure 10-9 using nested `while` loops:

```
var htmlCode = "<table>";
var rowNum = 1;
while (rowNum <= 2) {
   htmlCode += "<tr>";
   var colNum = 1;
   while (colNum <= 3) {
      htmlCode += "<td>" + rowNum + "," + colNum + "</td>";
      colNum++;
   }
   htmlCode += "</tr>";
   rowNum++;
}
```

Again, the initial values of the counter variables are set before the `while` loops are run and are updated within the command blocks.

TIP

Use a `for` loop when your loop contains a counter variable. Use a `while` loop for a more general stopping condition.

Because `for` loops and `while` loops share many of the same characteristics, which one you choose for a given application is often a matter of personal preference. In general, `for` loops are used whenever you have a counter variable and `while` loops are used for conditions that don't easily lend themselves to using counters. For example, you could construct a `while` loop that runs as long as the current time falls within a specified time interval.

Exploring the `do/while` Loop

In the `for` and `while` loops, the test to determine whether to continue the loop is made before the command block is run. JavaScript also supports a program loop called **`do/while`** that tests the condition to continue the loop right after the latest command block is run. The structure of the `do/while` loop is as follows:

```
do {
   commands
   }
while (continue);
```

For example, the following code is used to create the table shown earlier in Figure 10-8 as a `do/while` loop:

```
var htmlCode = "<tr>";
var i = 1;
do {
   htmlCode += "<td>" + i + "</td>";
   i++;
}
while (i <= 4);
htmlCode += "</tr>";
```

The `do/while` loop is usually used when the program loop should run at least once before testing for the stopping condition.

The <= symbol used in these program loops is an example of a comparison operator. Before continuing your work on the calendar app, you examine the different types of comparison operators supported by JavaScript.

Comparison and Logical Operators

A **comparison operator** is an operator that compares the value of one expression to another returning a Boolean value indicating whether the comparison is true or not. Thus, the following expression uses the < comparison operator to test whether the value of the `x` variable is less than 100:

```
x < 100
```

If this comparison is true, the expression returns the Boolean value `true` and, if otherwise, `false`. Figure 10-11 lists the comparison operators supported by JavaScript.

Figure 10-11 Comparison operators

Operator	Example	Description
==	`x == y`	Tests whether x is equal in value to y
===	`x === y`	Tests whether x is equal in value to y and has the same data type
!=	`x != y`	Tests whether x is not equal to y
>	`x > y`	Tests whether x is greater than y
>=	`x >= y`	Tests whether x is greater than or equal to y
<	`x < y`	Tests whether x is less than y
<=	`x <= y`	Tests whether x is less than or equal to y

When you want to test whether two values are equal, you use either a double equal sign (==) or a triple equal sign (===). The double equal sign tests whether two items are equal in value while the triple equal sign tests whether the two items are equal in value and also in data type. Thus, the following expression tests whether x is equal in value to 100 and is a number:

```
x === 100
```

Using the single equal sign (=) for the comparison operator is a common programming mistake; remember that the equal sign is an assignment operator and is reserved for setting one value equal to another, not for testing whether two values are equal.

JavaScript also supports **logical operators** that allow you to connect several expressions. For example, the logical operator `&&` returns a value of true only if both of the expressions are true. Figure 10-12 lists the JavaScript logical operators.

Figure 10-12 Logical operators

Operator	Definition	Example	Description
`&&`	and	`(x === 5) && (y === 8)`	Tests whether x is equal to 5 and y is equal to 8
`||`	or	`(x === 5) || (y === 8)`	Tests whether x is equal to 5 or y is equal to 8
`!`	not	`!(x < 5)`	Tests whether x is not less than 5

Program Loops and Arrays

Program loops can be used to cycle through the different values contained within an array. The general structure for accessing each value from an array using a `for` loop is

```
for (var i = 0; i < array.length; i++) {
   commands involving array[i]
}
```

where *array* is the array containing the values to be looped through and *i* is the counter variable used in the loop. The counter variable in this case represents the index number of an item from the array. The `length` property is used to determine the size of the array. The last item in the array has an index value of one less than the array's length—because array indices start with zero—so you continue the loop only when the array index is less than the length value.

REFERENCE

Creating Program Loops

- To create a `for` loop, use looping structure

```
for (start; continue; update) {
   commands
}
```

where *start* is an expression that sets the initial value of a counter variable, *continue* is a Boolean expression that must be true for the loop to continue, *update* is an expression that indicates how the value of the counter variable should change each time through the loop, and *commands* is the JavaScript commands that are run each time through the loop.

- To create a `while` loop, use the following structure:

```
while (continue) {
   commands
}
```

- To create a `do/while` loop, use the following:

```
do {
   commands
   }
while (continue);
```

- To loop through the contents of an array, enter the `for` loop

```
for (var i = 0; i < array.length; i++) {
   commands involving array[i]
}
```

where *i* is a counter variable representing the indices of the array items and *array* is the array to be looped through.

With this information, you can create a function that employs arrays and a `for` loop to create a row displaying the names of the seven days of the week. First, you will place the three-letter abbreviation of each weekday in an array and then loop through that array, writing a table heading cell for each day. You will place these commands in a function named calWeekdayRow().

To create the calWeekdayRow() function:

1. If you took a break after the previous session, make sure the **lht_calendar.js** file is open in your text editor.
2. At the bottom of the file, insert the following commands to begin creating the function by inserting an array named *dayName* containing the three-letter abbreviations of the seven days of the week:

```
/* Function to write a table row of weekday abbreviations */
function calWeekdayRow() {
   // Array of weekday abbreviations
   var dayName = ["SUN", "MON", "TUE", "WED", "THU", "FRI",
"SAT"];
```

3. Next, create the rowHTML variable containing the opening tag for the table row by inserting the following command:

```
var rowHTML = "<tr>";
```

You must enclose all commands in a `for` loop within a set of opening and closing curly braces so that each command is run every time through the loop.

4. Add the following `for` loop to loop through the contents of the dayName array, adding HTML code for each `th` element:

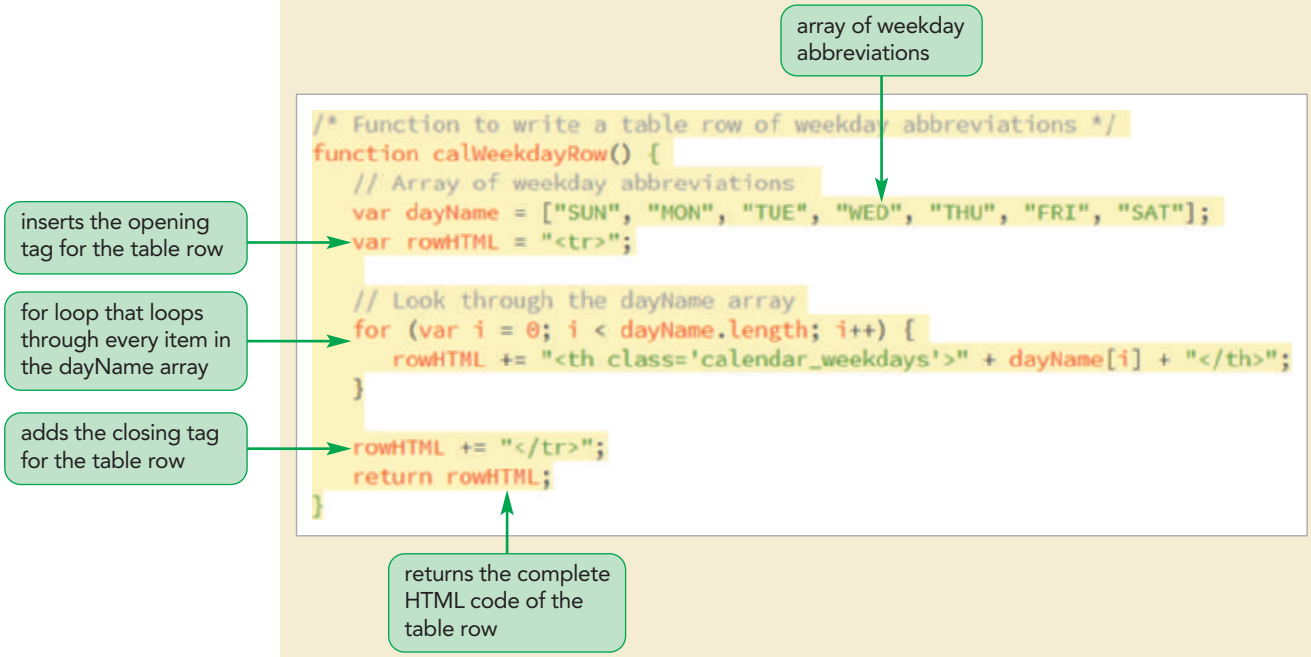
```
// Loop through the dayName array
for (var i = 0; i < dayName.length; i++) {
   rowHTML += "<th class='calendar_weekdays'>" + dayName[i] +
"</th>";
}
```

5. Finally, complete the calWeekdayRow() function by adding a closing </tr> tag to the value of the rowHTML variable and return that variable's value. Add the code that follows:

```
   rowHTML += "</tr>";
   return rowHTML;
}
```

Figure 10-13 shows the complete contents of the calWeekdayRow() function.

Figure 10-13 The calWeekdayRow() function

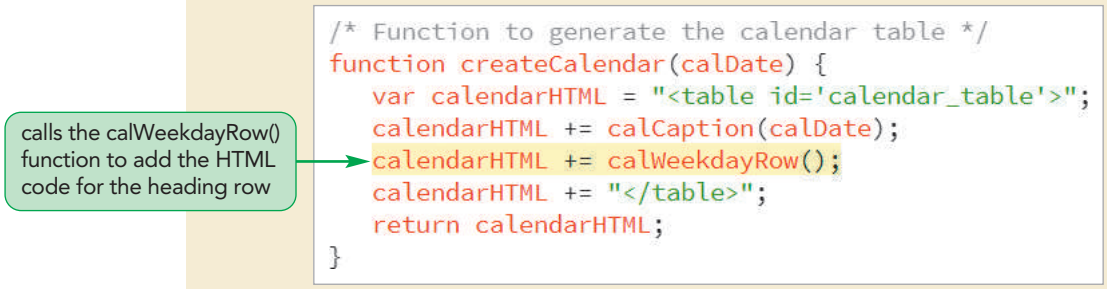



```
/* Function to write a table row of weekday abbreviations */
function calWeekdayRow() {
   // Array of weekday abbreviations
   var dayName = ["SUN", "MON", "TUE", "WED", "THU", "FRI", "SAT"];
   var rowHTML = "<tr>";

   // Look through the dayName array
   for (var i = 0; i < dayName.length; i++) {
      rowHTML += "<th class='calendar_weekdays'>" + dayName[i] + "</th>";
   }

   rowHTML += "</tr>";
   return rowHTML;
}
```

6. Scroll back up to the createCalendar() function and insert the following command as shown in Figure 10-14:

```
calendarHTML += calWeekdayRow();
```

Figure 10-14 Calling the calWeekdayRow() function

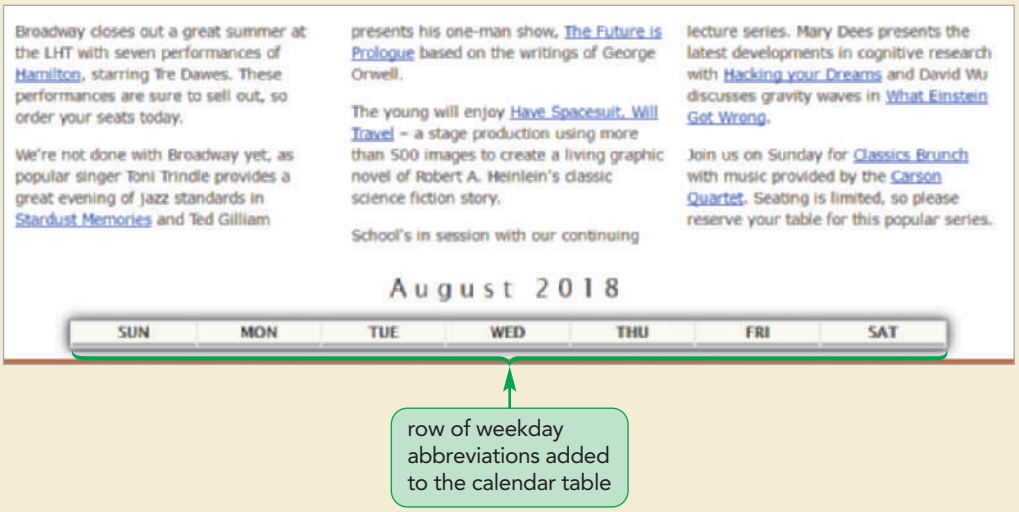



```
/* Function to generate the calendar table */
function createCalendar(calDate) {
   var calendarHTML = "<table id='calendar_table'>";
   calendarHTML += calCaption(calDate);
   calendarHTML += calWeekdayRow();
   calendarHTML += "</table>";
   return calendarHTML;
}
```

7. Save your changes to the file, and then reload the lht_august.html file in your browser.

   Figure 10-15 shows the revised appearance of the page with the calendar table now showing a row of weekday abbreviations.

Figure 10-15 Row of weekday abbreviations



INSIGHT

Returning a Random Array Item

In some programs, such as gaming apps, you might want to return a random value from an array. You can use the array index numbers along with the `Math.random` and `Math.floor` methods to achieve this. Assuming that an array is not sparse, the total number of array items is provided by the `length` property. To return a random index from the array, you use the expression

```
Math.floor(Math.random()*array.length);
```

where *array* is the name of the array. The value returned by this expression would be a random integer from 0 up to the value `length-1`, which corresponds to all of the array indices. You could place this expression in a function such as

```
function randItem(arr) {
   return arr[Math.floor(Math.random()*arr.length)];
}
```

using the `arr` parameter as the array to be evaluated.

To pick a random item from any array, you could apply the randItem() function to any array as follows

```
var color = ["red", "blue", "green", "yellow"];
var randColor = randItem(color);
```

and the randColor variable would contain one of the four colors chosen at random from the color array.

Array Methods to Loop Through Arrays

JavaScript supports several methods to loop through the contents of an array without having to create a program loop structure. Because these methods are built into the JavaScript language, they are faster than program loops; however, older browsers might not support them, so you should apply them with caution.

Each of these methods is based on calling a function that will be applied to each item in the array. The general syntax is

```
array.method(callback [, thisArg])
```

TIP

Callback function parameters can be given any descriptive name you choose, but the parameters must be listed in the order: value, index, and array name.

where *array* is the array, *method* is the array method, and *callback* is the name of the function that will be applied to each array item. An optional argument, *thisArg*, can be included to pass a value to the callback function. The general syntax of the callback function is

```
function callback(value [, index, array]) {
   commands
}
```

where *value* is the value of the array item during each pass through the array, *index* is the numeric index of the current array item, and *array* is the name of the array. Only the *value* parameter is required; the others are optional.

Running a Function for Each Array Item

The first method you will explore is `forEach()`, which is used to run a function for each item in the array. The general syntax is

```
array.forEach(callback [, thisArg])
```

where *callback* is the function that is applied to each item in the array. For example, the following `forEach()` method applies the sumArray() function with each item in the x array:

```
var sum = 0;
var x = [2, 5, 7, 12];

x.forEach(sumArray);

function sumArray(value) {
   sum += value;
}
```

Note that the sumArray() function has a single parameter named `value`, representing the current array item. The result of running the `forEach()` method with the sumArray() function is that the value of each item in the x array is added to the sum variable, resulting in a final value of 26 in this example. The `forEach()` method can also be used to modify the values of individual array items.

The following code calls the stepUp() function to increase the value of each item in the x array by 1:

```
var x = [4, 7, 11];

x.forEach(stepUp);

function stepUp(value, i, arr) {
   arr[i] = value + 1;
}
```

Notice that in this case, the stepUp() function has three parameters, with the second parameter (`i`) representing the array index and the third parameter (`arr`) representing the array itself. After running this code, the x array would contain the values [5, 8, 12].

Mapping an Array

The `map()` method performs an action similar to the `forEach()` method except that the function it calls returns a value that can be used to map the contents of an existing array into a new array. The following code demonstrates how to use the `map()`

method to create a new array in which each item is equal to twice the value of the corresponding item in the original array:

```
var x = [3, 8, 12];

var y = x.map(DoubleIt);

function DoubleIt(value) {
   return 2*value;
}
```

After running this code, the y array contains the values [6, 16, 24]. Note that the `map()` method does not affect the contents or structure of the original array, and the new array will have the same number of array items as the original. If the original array is sparse with several missing indices, the mapped array will have the same sparseness.

Filtering an Array

Often when working with arrays, you will want to extract array items that match some specified condition. For example, in an array of test scores, you might want to extract only those test scores with a value of 90 or above. The following `filter()` method can be used to create such arrays

```
array.filter(callback [, thisArg])
```

where *callback* is a function that returns a Boolean value of `true` or `false` for each item in the array. The array items that return a value of `true` get copied into the new array. The following code demonstrates how to use the `filter()` method to create a subarray of items whose value is greater than 90:

```
var scores = [92, 68, 83, 95, 91, 65, 77];

var highScores = scores.filter(gradeA);

function gradeA(value) {
   return value > 90;
}
```

After running this code, the highScores array would contain the values [92, 95, 91].

INSIGHT

Passing a Value to a CallBack Function

If you need to pass a value to a callback function used by any of the array methods, you can include the optional *thisArg* parameter. In the following code, a value of 92 is entered as argument in the `filter()` method in order to return array items whose value is greater than or equal to 92

```
var scores = [92, 68, 83, 95, 91, 65, 77];

var highScores = scores.filter(gradeA, 92);

function gradeA(value) {
   return value >= this;
}
```

resulting in an array with the values [92, 95]. Note that the gradeA function uses the JavaScript keyword `this` to represent the value of the *thisArg* parameter. The `this` keyword is an important part of the JavaScript language and is used to represent a current value being operated upon by the browser.

Another common use of arrays is to examine an array's contents to determine whether every array item satisfies a specified condition. The following `every()` method returns the value `true` if every item in the array matches the condition specified by the callback function and, if otherwise, returns `false`:

```
array.every(callback [, thisArg])
```

As with the `filter()` method, the function used by the `every()` method must return a Boolean value of `true` or `false`. For example, the following code uses the `every()` method to test whether every test score exceeds a value of 70:

```
var scores = [92, 68, 83, 95, 91, 65, 77];

var allPassing = scores.every(passTest);

function passTest(value) {
   return value > 70;
}
```

In this example, the value of the allPassing variable would be `false` because not every value in the scores array is greater than 70. Similarly, the following `some()` method

```
array.some(callback [,thisArg])
```

returns a value of `true` if some—but not necessarily all—array items match a condition specified in the function and a value of `false` if none of the array items match the condition specified in the function. Applying the `some()` method to the above array would return a value of true because some (but not all) of the scores are greater than 70.

Figure 10-16 summarizes the different JavaScript array methods that can be used to work with the collection of items within an array.

Figure 10-16 Array methods to loop through arrays

Array Method	Description
every(*callback* [, *thisArg*])	Tests whether the condition returned by the *callback* function holds for all items in *array*; in all array methods, the optional *thisArg* parameter is used to pass values to the *callback* function
filter(*callback* [, *thisArg*])	Creates a new array populated with the elements of *array* that return a value of *true* from the *callback* function
forEach(*callback* [, *thisArg*])	Applies the *callback* function to each item in *array*
map(*callback* [, *thisArg*])	Creates a new array by passing the original array items to the *callback* function, which returns the mapped value of the array items
reduce(*callback* [, *thisArg*])	Reduces *array* by keeping only those items that return a value of *true* from the *callback* function
reduceRight(*callback* [, *thisArg*])	Reduces *array* from the last element by keeping only those items that return a value of *true* from the *callback* function
some(*callback* [, *thisArg*])	Tests whether the condition returned by the *callback* function holds for at least one item in *array*
find(*callback* [, *thisArg*])	Returns the value of the first element in the array that passes a test in the *callback* function
findIndex(*callback* [, *thisArg*])	Returns the index of the first element in the array that passes a test in the *callback* function

PROSKILLS

Decision Making: Efficient Loops

As your programs increase in size and complexity, the ability to write efficient code becomes essential. Bloated, inefficient code is particularly noticeable with program loops that might repeat the same set of commands hundreds or thousands of times. A millisecond wasted due to one poorly written command can mean an overall loss of dozens of seconds when it is part of a loop. Because studies show that users will rarely wait more than a few seconds for program results, it is important to shave off as many milliseconds as you can. Here are some ways to speed up your loops:

- ***Calculate outside the loop***. There is no reason to repeat the exact same calculation hundreds of times within a loop. For example, the following code unnecessarily recalculates the same Math.log(cost) value a thousand times in the `for` loop:

```
for (i = 0; i < 1000; i++) {
   x[i] = i*Math.log(cost);
}
```

Instead, place that calculation outside the loop, where it will be calculated only once:

```
var costLog = Math.log(cost);
for (i = 0; i < 1000; i++) {
   x[i] = i*costLog;
}
```

- ***Determine array lengths once***. Rather than forcing JavaScript to count up the length of a large array each time through the loop, calculate the length before the loop starts:

```
var x = myArray.length;
for (var i = 0; i < x; i++) {
   commands
}
```

- ***Decrement rather than increment***. Instead of counting up to an array length, count down from the array length to 0, as in the following `for` loop:

```
var x = myArray.length;
for (var i = x; i--) {
   commands
}
```

When the counter variable equals 0, the loop will stop.

- ***Unroll the loop***. When only a few items are being iterated in a loop, it is actually faster not to use a program loop. Instead, enter each counter value explicitly in separate statements.
- ***Manage Loop size***: A long command block is a red flag warning you that you might be trying to do too much each time through a loop. Look for ways to reduce the number of tasks and calculations in the command block to a bare minimum.

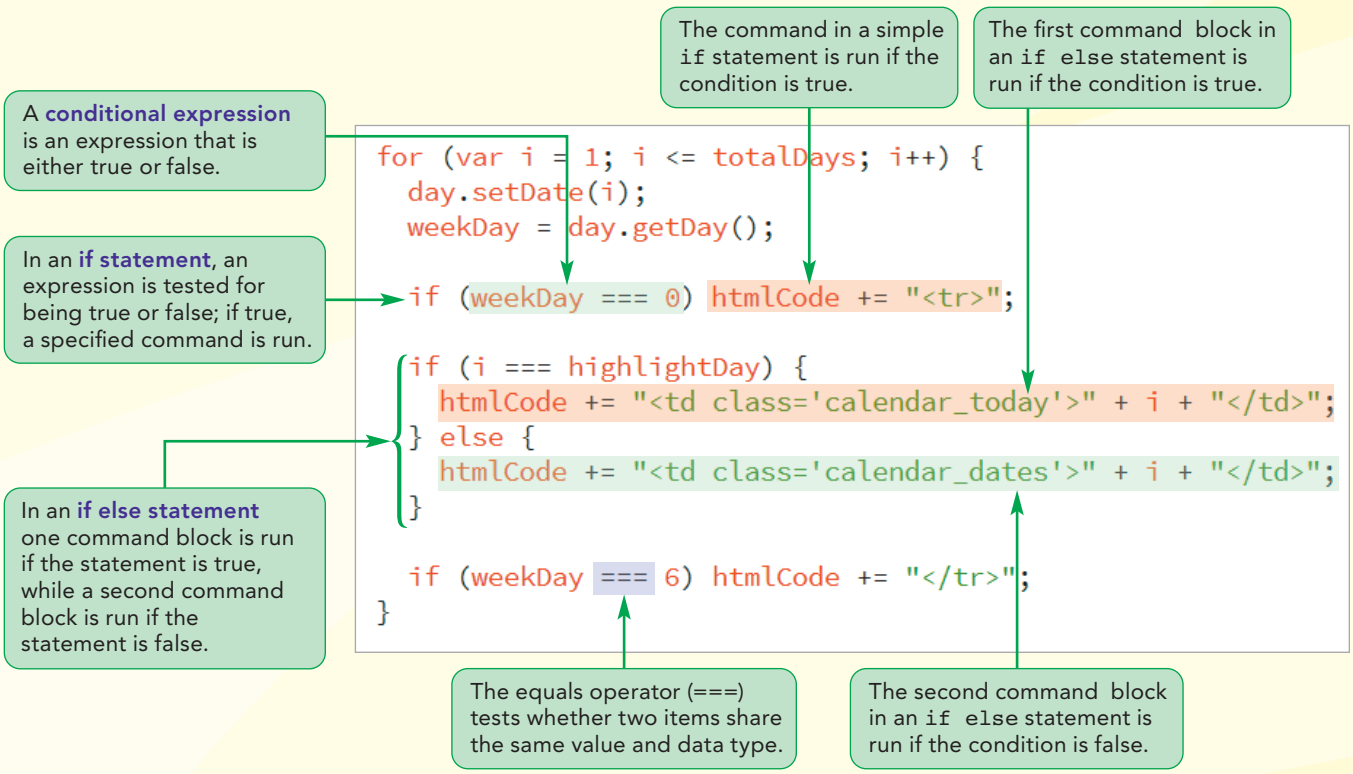
In the next session, you will explore how to work with JavaScript's conditional statements and put together everything you have learned to complete the calendar app.

REVIEW

Session 10.2 Quick Check

1. What is a program loop? Name three types of program loops supported by JavaScript.
2. Provide a `for` statement to use a counter variable named i that starts with the value 0 and continues up to 100 in increments of 10.
3. Provide a `for` statement that stores the HTML code for a table row consisting of five table cells in a variable named tableCode. Assume the table cells display the text `Column i`, where *i* is the value of the counter variable, and the value of the counter variable increases from 1 to 5 in increments of 1.
4. Provide code to duplicate the task in the previous question using a while loop.
5. Provide code, using an array method, to increase the value of each item in the array x = [2, 14, -3, 7] by 10.
6. Provide code, using an array method, to map the value of items in the array x = [2, 14, -3, 7] into a new array named y in which each of the values is increased by 10.
7. Provide code, using an array method, to return a Boolean value indicating whether every value in the array x = [2, 14, -3, 7] is positive.
8. Provide code, using an array method, to store only the positive values from the array x = [2, 14, -3, 7] in a new array named y.

Session 10.3 Visual Overview:



Conditional Statements

```
function daysInMonth(calDate) {
   var dayCount = [31,28,31,30,31,30,31,31,30,31,30,31];

   var thisYear = calDate.getFullYear();
   var thisMonth = calDate.getMonth();

   if (thisYear % 4 === 0) {
      if ((thisYear % 100 != 0) || (thisYear % 400 === 0)) {
         dayCount[1] = 29;
      }
   }

   return dayCount[thisMonth];

}
```

The or operator (||) is used when either of two conditions may be true for the entire conditional expression to be true.

In a nested if structure, one `if` statement is placed within another; the nested `if` statement is run only if the conditional expressions of both the outer and inner `if` statements are true.

Introducing Conditional Statements

Your next task in your calendar app is to create a program loop that writes the days of the month, entered within different table cells arranged in separate table rows. The process should end when the last day of the month is reached. Because months have different numbers of days, you first need to create a function named daysInMonth() that determines the number of days in a given month.

Like the calCaption() function you created earlier, the daysInMonth() function will have a single parameter, calDate, containing a `Date` object on which your calendar will be based. The function will also store the year value and month value in the variables thisYear and thisMonth, respectively, and will contain the following array that stores the number of days in each month:

```
var dayCount = [31,28,31,30,31,30,31,31,30,31,30,31];
```

This array is an example of a **parallel array** because each entry in the array matches—or is parallel to—an entry in the monthName array you created in the first session. To return the days of the month from the calendar date, the function will use the value of the thisMonth variable to reference the corresponding day value in the dayCount array with the following expression:

```
dayCount[thisMonth]
```

So, for instance, given the date July 6, 2018, the function would return the value 31.

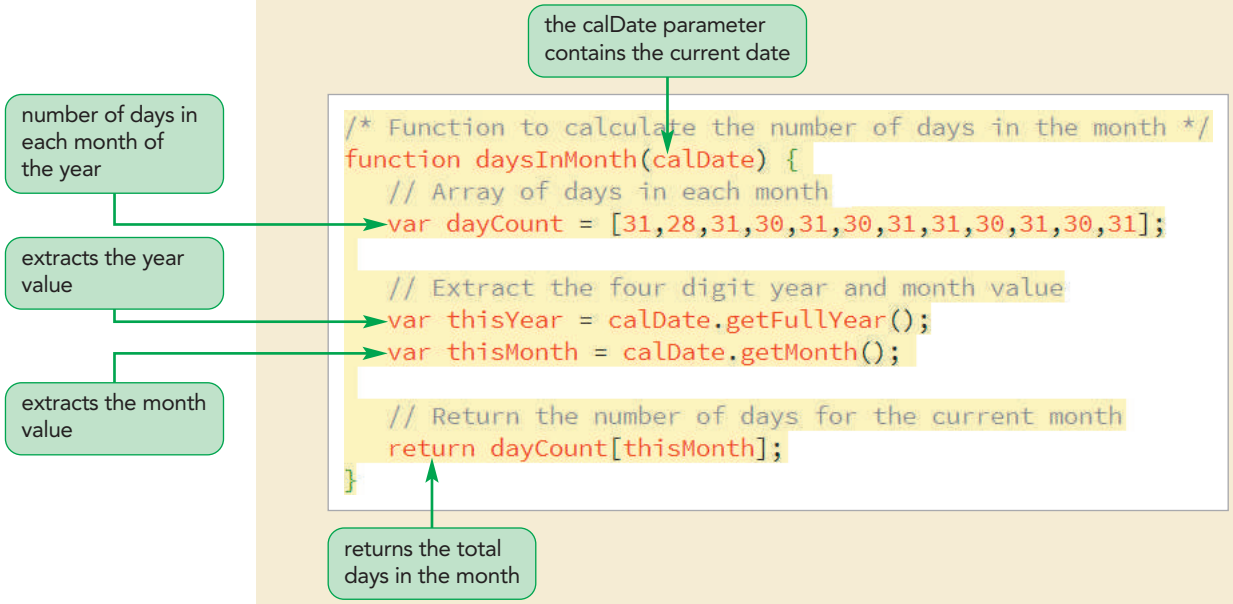
You add the daysInMonth() function now.

To start creating the daysInMonth() function:

1. If you took a break after the previous session, make sure the **lht_calendar.js** file is open in your text editor.
2. At the bottom of the file, insert the following code, as shown in Figure 10-17:

```
* Function to calculate the number of days in the month */
function daysInMonth(calDate) {
   // Array of days in each month
   var dayCount = [31,28,31,30,31,30,31,31,30,31,30,31];

   // Extract the four digit year and month value
   var thisYear = calDate.getFullYear();
   var thisMonth = calDate.getMonth();

   // Return the number of days for the current month
   return dayCount[thisMonth];
}
```

Figure 10-17 **Inserting the daysInMonth() function**



3. Save your changes to the file.

Perhaps you have already noticed a problem with the dayCount array: February has 29 days during a leap year, not 28 days as shown in the array. For the daysInMonth() function to return the correct value for the month of February, it must examine the year value and then set the value for the number of days in February to either 28 or 29 based on whether the current year is a leap year. You can do this through a conditional statement. A **conditional statement** is a statement that runs a command or command block only when certain circumstances are met.

Exploring the `if` Statement

The most common conditional statement is the `if` statement, which has the structure

```
if (condition) {
   commands
}
```

where *condition* is a Boolean expression that is either true or false, and *commands* is the command block that is run if *condition* is true. If only one command is run, you can eliminate the command block and enter the `if` statement as follows:

```
if (condition) command;
```

A conditional statement uses the same comparison and logical operators you used with the program loops in the last session. For example, the following `if` statement would set the value of the dayCount array for February to 29 if the year value were 2020 (a leap year):

```
if (thisYear === 2020) {
   dayCount[1] = 29;
}
```

For the calendar app, you will need to create a conditional expression that tests whether the current year is a leap year and then sets the value of dayCount[1]

appropriately. The general rule is that leap years are divisible by 4, so you will start by looking at operators that can determine whether the year is divisible by 4. One way is to use the `%` operator, which is also known as the modulus operator. The **modulus operator** returns the integer remainder after dividing one integer by another. For example, the expression `15 % 4` returns the value 3 because 3 is the remainder after dividing 15 by 4. To test whether a year value is divisible by 4, you use the conditional expression

```
thisYear % 4 === 0
```

where the thisYear variable contains the four-digit year value. The following is the complete `if` statement to change the value of the dayCount array for the month of February:

```
if (thisYear % 4 === 0) {
   dayCount[1] = 29;
}
```

Add this `if` statement to the daysInMonth() function now.

To revise the daysInMonth() function:

Be sure to use the triple equal sign symbol (===) and not the single equal sign symbol (=) when making a comparison in an `if` statement.

1. After the statement that declares the thisMonth variable, insert the following `if` statement:

```
// Revise the days in February for leap years
if (thisYear % 4 === 0) {
   dayCount[1] = 29;
}
```

Figure 10-18 highlights the newly added code in the function.

Figure 10-18 Inserting an if statement

tests whether thisYear is evenly divisible by 4

if it is, sets the value of dayCount[1] (February) to 29

```
/* Function to calculate the number of days in the month */
function daysInMonth(calDate) {
   // Array of days in each month
   var dayCount = [31,28,31,30,31,30,31,31,30,31,30,31];

   // Extract the four digit year and month value
   var thisYear = calDate.getFullYear();
   var thisMonth = calDate.getMonth();

   // Revise the days in February for leap years
   if (thisYear % 4 === 0) {
      dayCount[1] = 29;
   }

   // Return the number of days for the current month
   return dayCount[thisMonth];

}
```

2. Save your changes to the file.

INSIGHT

Assigning Values with Conditional Operators

When you want to simply assign a value to a variable rather than run a command block, you can write a more compact conditional expression using a **conditional operator** or a **ternary operator**, which has the syntax

```
condition ? value1 : value2;
```

where *condition* is a Boolean expression, *value1* is the value if the expression is true and *value2* is the value if the expression is false. For example, the following statement assigns a value of "Morning" to the session variable if the hour variable is less than 12 and "Afternoon" if otherwise:

```
var session = hour < 12 ? "Morning" : "Afternoon";
```

Conditional operators can test more than one possible condition by adding a second conditional operator to the last term in the expression as follows

```
condition1 ? value1 : condition2 ? value2 : value3;
```

where *value1* is assigned if *condition1* is true, *value2* is assigned if *condition2* is true (but not *condition1*), and *value3* is assigned if neither *condition1* nor *condition2* are true. Thus, the following statement assigns one of three possible values to the session variable based on the value of the hour variable:

```
var session = hour < 12 ? "Morning" : hour < 16 ? "Afternoon" :
"Evening";
```

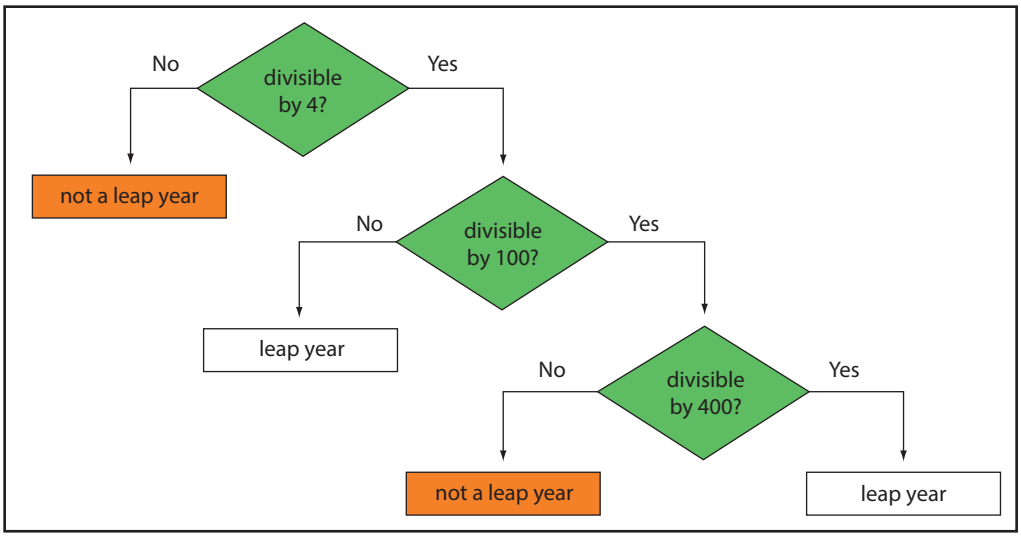
If hour is less than 12, the session variable has the value "Morning"; if hour is less than 16 (but greater than 12), the value is "Afternoon"; and otherwise, the value of the session variable is "Evening".

Note that conditional operators can only be used to assign a value. If you need to do more than one action in response to a conditional expression, use an `if` statement.

Nesting `if` Statements

The `if` statement you wrote for the daysInMonth() function works as a simple approximation, but it is not completely accurate. In most cases, a year that is evenly divisible by 4 is a leap year. The only exceptions are years that occur at the turn of the century, which are evenly divisible by 100. These years are not leap years unless they are also evenly divisible by 400. Thus, years such as 1800, 1900, and 2100 are not leap years even though they are evenly divisible by 4. Years such as 2000 and 2400 are leap years because they are evenly divisible by 400. Figure 10-19 shows the complete process used to determine whether a particular year is a leap year.

Figure 10-19 Process to calculate leap years



To translate these rules into our calendar app, you need to nest one `if` statement inside another one. The general structure of this nested `if` statement is as follows:

```
if (thisYear % 4 === 0) {
   further test for century years
}
```

The nested `if` statement needs to add two more conditions: (1) the year is not divisible by 100, and (2) the year is divisible by 400. The expressions for these two conditions are as follows:

```
thisYear % 100 != 0
thisYear % 400 === 0
```

If either of those two conditions is true for a year evenly divisible by 4, then the year is a leap year. Note that you will use the not equal to operator (`!=`) to test for an inequality in the first expression. You will then combine these two expressions into a single expression using the `or` operator (`||`), as follows:

```
(thisYear % 100 != 0) || (thisYear % 400 === 0)
```

Finally, you will nest this conditional expression as follows:

```
if (thisYear % 4 === 0) {
   if ((thisYear % 100 != 0) || (thisYear % 400 === 0)) {
      dayCount[1] = 29;
   }
}
```

Under this set of nested `if` statements, the number of days in February is 29 only if the thisYear variable is divisible by 4, and then only if it is also divisible by 400 or not divisible by 100. Take some time to compare this set of nested `if` statements with the chart shown earlier in Figure 10-19 to confirm that it satisfies all possible conditions for leap years. After incorporating this set of nested `if` statements, the daysInMonth() function returns the correct number of days for any month in any given year.

To complete the daysInMonth() function:

1. Within the `if` statement you entered in the last set of steps, delete the statement

```
dayCount[1] = 29
```

2. Replace the statement you just deleted with the following nested `if` statement:

```
if ((thisYear % 100 != 0) || (thisYear % 400 === 0)) {
   dayCount[1] = 29;
}
```

Figure 10-20 highlights the newly inserted nested `if` statement.

Figure 10-20 **Inserting a nested if statement**

if the year is divisible by 4 and either not divisible by 100 or divisible by 400, it's a leap year

```
// Revise the days in February for leap years
if (thisYear % 4 === 0) {
   if ((thisYear % 100 != 0) || (thisYear % 400 === 0)) {
      dayCount[1] = 29;
   }
}
```

3. Save your changes to the file.

Exploring the `if else` Statement

The `if` statement runs a command or a command block only if the conditional expression returns the value `true`; it does nothing if the condition is false. On some occasions, you might want to choose between alternate command blocks so that one command block is run if the conditional expression is true, and a different command block is run if the expression is false. The general structure of an `if else` statement follows:

```
if (condition) {
   commands if condition is true
} else {
   commands if condition is false
}
```

If only a single command is run in response to the `if` statement, you can use the following abbreviated form:

```
if (condition) command if condition is true
else command if condition is false;
```

The following example shows an `if else` statement that displays two possible alert boxes depending on whether the value of the day variable is Friday or not:

```
if (day === "Friday") alert("Thank goodness it's Friday")
else alert("Today is " + day);
```

Like `if` statements, `if else` statements can be nested as in the following code, which chooses between three possible alert boxes:

```
if (day === "Friday") alert("Thank goodness it's Friday")
else {
   if (day === "Monday") alert("Blue Monday")
   else alert("Today is " + day);
}
```

TIP

To make it easier to interpret nested `if` statements, always indent your code, lining up all of the commands for one set of nested statements.

Some programmers advocate always using curly braces even if the command block contains only a single command. This practice visually separates one `else` clause from another. Also, when reading through nested statements, it can be helpful to remember that an `else` clause usually pairs with the nearest preceding `if` statement.

Using Multiple `else if` Statements

For more complex scripts, you might need to choose from several alternatives. In these cases, you can specify multiple `else` clauses, each with its own `if` statement. This is not a new type of conditional structure, but rather a way of taking advantage of the syntax rules inherent in the `if else` statement. The general structure for choosing from several alternatives is

```
if (condition1) {
   commands1
}  else if (condition2) {
   commands2
}  else if (condition3) {
   commands3
...
}  else {
   default commands
}
```

TIP

To simplify code, keep your nesting of multiple if statements to three or less, if possible. For more conditions, use the `case/switch` structure.

where *condition 1*, *condition 2*, *condition 3*, and so on are the different conditions to be tested. This construction should always include a final `else` clause that is run by default if none of the preceding conditional expressions is true. When a browser runs a series of statements like this one, it stops examining the remaining `else` clauses at the first true condition. The structure in the following example employs multiple `else if` conditions:

```
if (day === "Friday") {
   alert("Thank goodness it's Friday");
}  else if (day === "Monday") {
   alert("Blue Monday");
}  else if (day === "Saturday") {
   alert("Sleep in today");
}  else {
   alert("Today is " + day);
}
```

REFERENCE

Working with Conditional Statements

- To test a single condition, use the construction

```
if (condition) {
   commands
}
```

where *condition* is a Boolean expression and *commands* is a command block run if the conditional expression is true.

- To test between two conditions, use the following construction:

```
if (condition) {
   commands if condition is true
}  else {
   commands if not true
}
```

- To test multiple conditions, use the construction

```
if (condition1) {
   commands1
}  else if (condition2) {
   commands2
}  else if (condition3) {
   commands3
…
}  else {
   default commands
}
```

where *condition 1*, *condition 2*, *condition 3*, and so on are the different conditions to be tested. If no conditional expressions return the value true, the *default command block* is run.

You now have all of the tools you need to complete the calendar app. The only remaining task involves writing out the table cells containing the calendar days so that they are organized into separate rows. You will complete the calendar app in the next section.

INSIGHT

Exploring the switch Statement

Another way to handle multiple conditions is with the **switch statement**—also known as the **case statement**—in which different commands are run based upon different possible values of a specified variable. The syntax of the `switch` statement is

```
switch (expression) {
   case label1: commands1; break;
   case label2: commands2; break;
   case label3: commands3; break;
   ...
   default: default commands
}
```

where *expression* is an expression that returns a value; *label1*, *label2*, and so on are possible values of that expression; *commands1*, *commands2*, and so on are the commands associated with each label; and *default commands* is the set of commands to be run if no label matches the value returned by expression. The following `switch` statement demonstrates how to display a different alert box based on the value of the day variable:

```
switch (day) {
   case "Friday": alert("Thank goodness it's Friday"); break;
   case "Monday": alert("Blue Monday"); break;
   case "Saturday": alert("Sleep in today"); break;
   default: alert("Today is " + day);
}
```

The `break` statement is optional and is used to halt the execution of the `switch` statement once a match has been found. For programs with multiple matching cases, you can omit the `break` statements and JavaScript will continue moving through the switch statements, running all matching commands.

Because of its simplicity, the switch statement is often preferred over a long list of `else if` statements that can be confusing to read and to debug.

Completing the Calendar App

The last part of creating the calendar involves writing table cells for each day of the month. The completed calendar app must do the following:

- Calculate the day of the week in which the month starts.
- Write blank table cells for the days before the first day of the month.
- Loop through the days of the current month, writing each date in a different table cell and starting a new table row on each Sunday.

You will place all of these commands in a function named calDays(). The function will have a single parameter named calDate storing a `Date` object for the current date. You add this function to the lht_calendar.js file.

To start the calDays() function:

1. At the bottom of the lht_calendar.js file, insert the following function:

```
/* Function to write table rows for each day of the month */
function calDays(calDate) {
   // Determine the starting day of the month

   // Write blank cells preceding the starting day

   // Write cells for each day of the month
}
```

Figure 10-21 highlights the initial code of the function, as well as comments to help explain the code that will be added.

Figure 10-21 **Inserting the calDays() function and comments**

```
/* Function to write  table rows for each day of the month */
function calDays(calDate) {
   // Determine the starting day of the month

   // Write blank cells preceding the starting day

   // Write cells for each day of the month
}
```

2. Save your changes to the file.

Setting the First Day of the Month

To loop through all of the days of the month, you need to keep track of each day as its table cell is written into the calendar table. You will store this information in a `Date` object named day. The initial value of the day variable will be set to match the first day of the calendar month using the following expression:

```
var day = new Date(calDate.getFullYear(), calDate.getMonth(), 1);
```

Note that the new Date() object constructor uses the four-digit year value and month value from the calDate parameter to set the year and month, and then sets the day value to 1 to match the first day of the month. For example, if the current date is August 12, 2017, the date stored in the day variable will be August 1, 2017; that is, no matter what current day is, the date stored in the day variable will be the first day for that month and year.

Next, to determine the day of the week on which the month starts, you use the following `getDay()` method:

```
var weekDay = day.getDay();
```

Recall that the `getDay()` method returns an integer ranging from 0 (Sunday) to 6 (Saturday). You add these two commands to the calDays() function now.

To create the day and weekDay variables:

1. Below the first comment in the calDays() function, insert the following commands:

```
var day = new Date(calDate.getFullYear(), calDate.getMonth(), 1);
var weekDay = day.getDay();
```

Figure 10-22 highlights the newly added code.

Figure 10-22 Calculating the start day of the month

sets the first day of the month

determines the weekday on which the month begins

```
/* Function to write  table rows for each day of the month */
function calDays(calDate) {
   // Determine the starting day of the month
   var day = new Date(calDate.getFullYear(), calDate.getMonth(), 1);
   var weekDay = day.getDay();

   // Write blank cells preceding the starting day

   // Write cells for each day of the month
}
```

2. Save your changes to the file.

Placing the First Day of the Month

Before the first day of the month, the calendar table should show only empty table cells that represent the days from the previous month. The value of the weekDay variable indicates how many empty table cells you need to create. For example, if the value of the weekDay variable is 4, indicating that the month starts on a Thursday, you know that there are four blank table cells—corresponding to Sunday, Monday, Tuesday, and Wednesday—that need to be written at the start of the first table row. The following loop writes the HTML code for the empty table cells to start the table row:

```
var htmlCode = "<tr>";
for (var i = 0; i < weekDay; i++) {
   htmlCode += "<td></td>";
}
```

Note that if weekDay equals 0—indicating that the month starts on a Sunday—then no blank table cells will be written because the value of the counter variable is never less than the value of the weekDay variable and thus, the command block in the `for` loop is completely skipped.

To write the initial blank cells of the first table row:

1. Below the second comment line, insert the following `for` loop:

```
var htmlCode = "<tr>";
for (var i = 0; i < weekDay; i++) {
   htmlCode += "<td></td>";
}
```

Figure 10-23 highlights the code for the `for` loop.

Figure 10-23 **Inserting blank cells for the days that precede the start of the month**

inserts opening <tr> tag for the initial table row

inserts a blank table cell for each weekday prior to the first of the month

```
/* Function to write  table rows for each day of the month */
function calDays(calDate) {
   // Determine the starting day of the month
   var day = new Date(calDate.getFullYear(), calDate.getMonth(), 1);
   var weekDay = day.getDay();

   // Write blank cells preceding the starting day
   var htmlCode = "<tr>";
   for (var i = 0; i < weekDay; i++) {
      htmlCode += "<td></td>";
   }

   // Write cells for each day of the month
}
```

2. Save your changes to the file.

Writing the Calendar Days

Finally, you will write the table cells for each day of the month using the following `for` loop:

```
var totalDays = daysInMonth(calDate);

for (var i = 1; i <= totalDays; i++) {
   day.setDate(i);
   weekDay = day.getDay();

   if (weekDay === 0) htmlCode += "<tr>";
   htmlCode += "<td class='calendar_dates'>" + i + "</td>";
   if (weekDay === 6) htmlCode += "</tr>";
}
```

The code starts by determining the total days in the month using the daysInMonth() function you created earlier. It then loops through those days, and each time through the loop it changes the day and weekDay variables to match the current day being written. If the day is a Sunday, a new table row is started; if the day is a Saturday, the current table row is ended. Each table cell displays the day number and belongs to the calendar_dates class, which allows it to be styled using the style rule from the lht_calendar.css style sheet.

To write the calendar days:

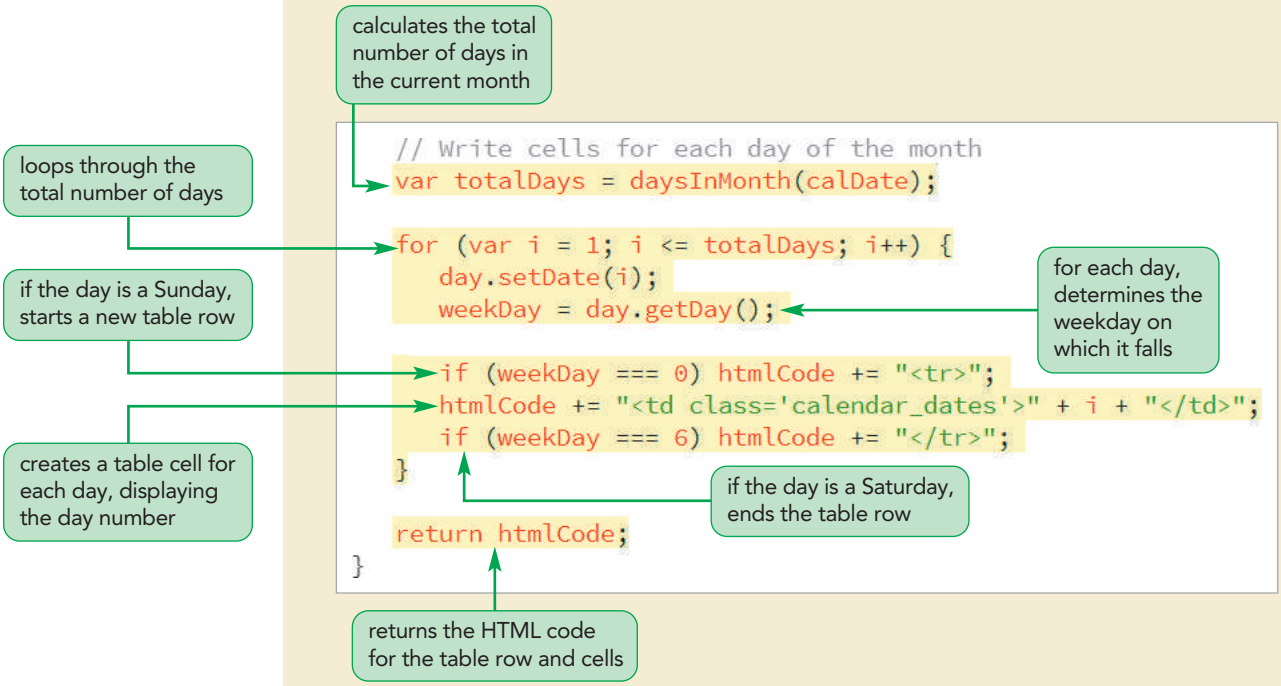
1. Below the last comment in the calDays() function, add the following commands:

```
var totalDays = daysInMonth(calDate);

for (var i = 1; i <= totalDays; i++) {
   day.setDate(i);
   weekDay = day.getDay();
```

```
    if (weekDay === 0) htmlCode += "<tr>";
    htmlCode += "<td class='calendar_dates'>" + i + "</td>";
    if (weekDay === 6) htmlCode += "</tr>";
  }

  return htmlCode;
```

Figure 10-24 highlights the code to write the table cells for each day of the month.

Figure 10-24 Writing the HTML code for the table row and cells



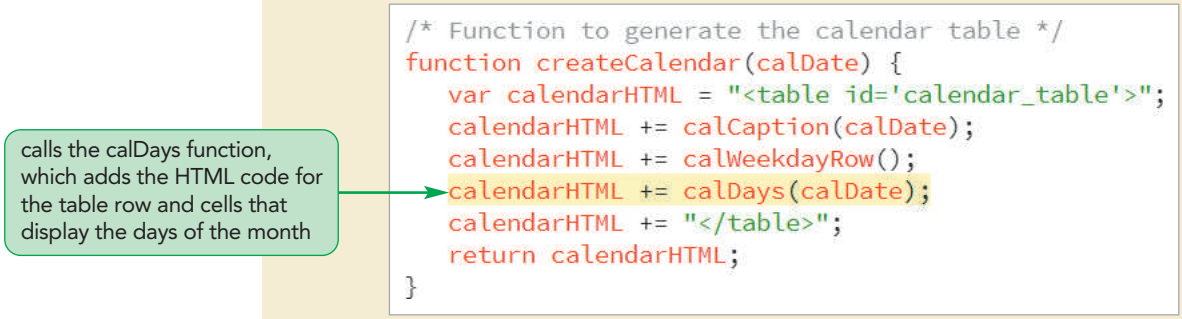
Next, you call the calDays() function from within the createCalendar() function and view the results.

2. Scroll up to the createCalendar() function, and then insert the following statement directly above the command that writes the closing `</table>` tag.

```
calendarHTML += calDays(calDate);
```

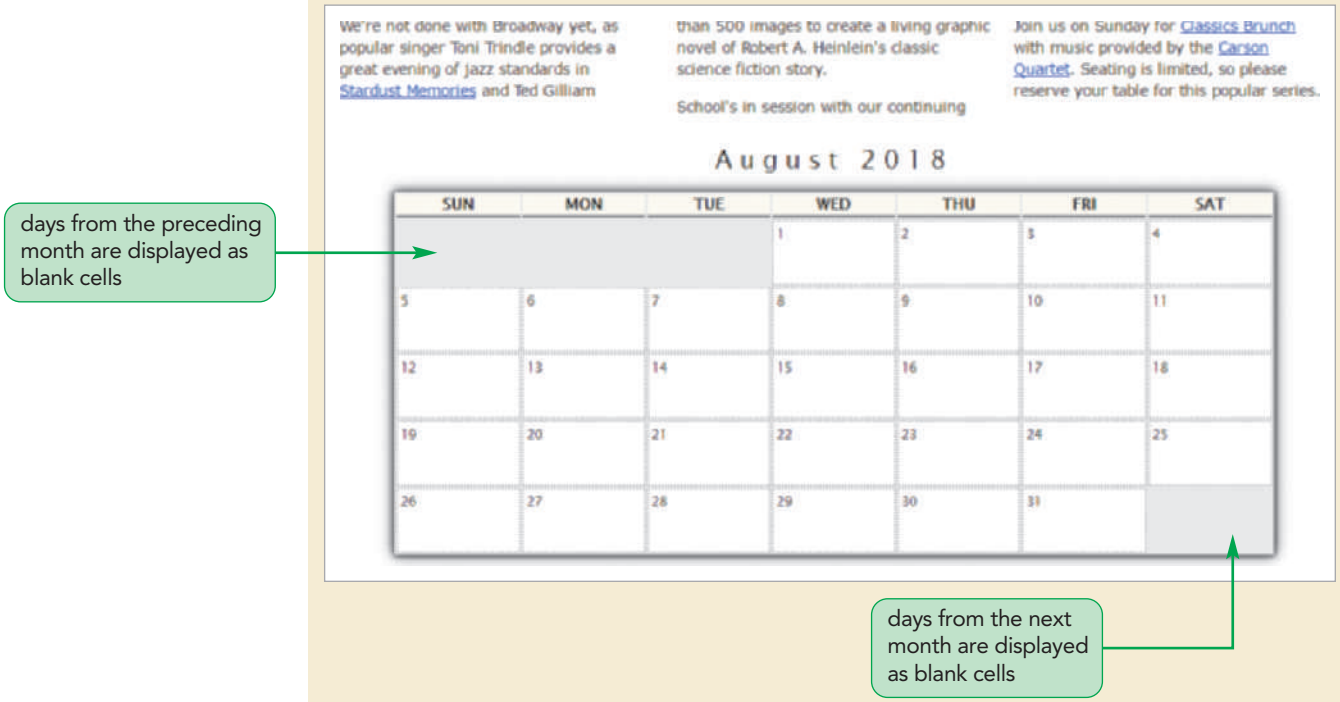
Figure 10-25 highlights the code in the function.

Figure 10-25 Calling the calDays() function



3. Save your changes to the file, and then reload the lht_august.html file in your browser. As shown in Figure 10-26, the page should now display the monthly calendar for August, 2018.

Figure 10-26 **Monthly calendar for August, 2018**



Trouble? If you do not see a calendar, you might have made a mistake in the code. Common mistakes include misspelling variable names, forgetting to close quoted text strings, inconsistently using uppercase and lowercase letters in variable names, and omitting closing braces in command blocks. Compare your code to the complete code of the calDays() function shown in Figures 10-23 and 10-24.

Highlighting the Current Date

Lewis likes the calendar's appearance but mentions that the calendar should also highlight the current day: August 24, 2018. Recall that Lewis has created a special style rule for the current day, identified using the HTML `id` value "calendar_today". Thus, to highlight that table cell, the calDays() function should test each day as it is being written; and if the date matches the calendar day, the function should write the table cell as

```
<td class='calendar_dates' id='calendar_today'>day</td>
```

where *day* is the day number. Otherwise, the function should write the table cell without the `id` attribute as follows:

```
<td class='calendar_dates'>day</td>
```

To determine the day number of the calendar day, you create the highlightDay variable, using the `getDate()` method to extract the day value from the calDate parameter. When the counter in the `for` loop matches the value of this variable, the loop will write the table cell including the calendar_today `id` attribute.

To highlight the current date in the calendar:

1. Return to the **lht_calendar.js** file in your editor, and then scroll down to the calDays() function.
2. In the Write cells for each day of the month section and directly above the `for` loop in that section, insert the following statement to calculate the day value of the current day:

```
var highlightDay = calDate.getDate();
```

3. Replace the statement that writes the table cell in the `for` loop with the following code:

```
if (i === highlightDay) {
   htmlCode += "<td class='calendar_dates' id='calendar_
today'>" + i + "</td>";
} else {
   htmlCode += "<td class='calendar_dates'>" + i + "</td>";
}
```

Figure 10-27 highlights the newly added `if` statement in the function.

TIP

Calculations such as the `getDate()` method that need to be performed once should always be placed outside the program loop to avoid unnecessarily repeating the same calculation each time through the loop.

Figure 10-27 Highlighting the current date in the calendar

stores the current day in the highlightDay variable

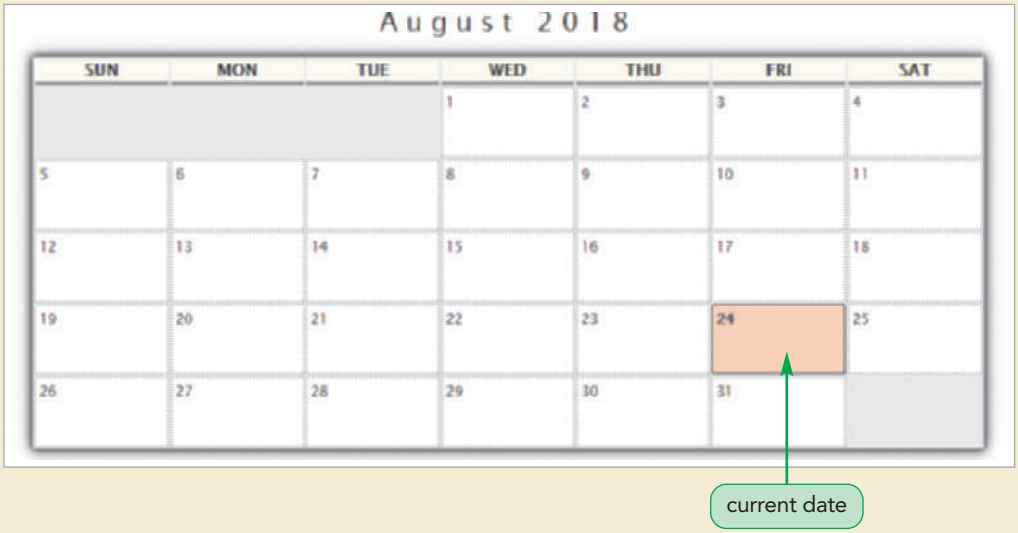
if the day is the highlight day, write a table cell with the id 'calendar_today'

otherwise write a table cell with no id value

```
   // Write cells for each day of the month
   var totalDays = daysInMonth(calDate);

   var highlightDay = calDate.getDate();
   for (var i = 1; i <= totalDays; i++) {
      day.setDate(i);
      weekDay = day.getDay();

      if (weekDay === 0) htmlCode += "<tr>";
      if (i === highlightDay) {
         htmlCode += "<td class='calendar_dates' id='calendar_today'>" + i + "</td>";
      } else {
         htmlCode += "<td class='calendar_dates'>" + i + "</td>";
      }
      if (weekDay === 6) htmlCode += "</tr>";
   }

   return htmlCode;
}
```

4. Save your changes to the file, and then reload lht_august.html in your browser. The table cell corresponding to August 24, 2018 should now be highlighted as shown in Figure 10-28.

Figure 10-28 Calendar with the current date highlighted



Displaying Daily Events

The final piece of your calendar app is to display the daily events in August. Lewis already has created an array of daily event text, part of which is shown in Figure 10-29.

Figure 10-29 The dayEvent array

```
var dayEvent = new Array();

dayEvent[1] = "";
dayEvent[2] = "<br /><a href='#'>Classic Cinema: Wings</a><br />7 pm<br />$5";
dayEvent[3] = "<br /><a href='#'>The Future is Prologue</a><br />8 pm<br />$18/$24/$36";
dayEvent[4] = "<br /><a href='#'>American Favorites</a><br />7:30 pm<br />$24/$36/$48";
dayEvent[5] = "<br /><a href='#'>Classics Brunch</a><br />11 am<br />$12";
dayEvent[6] = "<br /><a href='#'>LHT Jazz Band</a><br />7 pm<br />$24";
dayEvent[7] = "";
```

The dayEvent array has 31 items to match the 31 days in August. Array items that match days on which no event is scheduled contain a blank text string, while daily events are written in the HTML code that will be inserted into the calendar table. To display this content, you create a link to the lht_events.js file and then, within the calDays() function, you add an expression to write the contents of the dayEvent array into the individual table cells.

To display the daily events:

1. Return to the **lht_august.html** file in your text editor. Directly above the `script` element for the lht_calendar.js file, insert the following `script` element for the lht_events.js file:

   ```
   <script src="lht_events.js" defer></script>
   ```

   Figure 10-30 highlights the newly added code.

Figure 10-30 Linking to the lht_events.js file

links to the script file containing the dayEvents array

```
    <link href="lht_base.css" rel="stylesheet" />
    <link href="lht_layout.css" rel="stylesheet" />
    <link href="lht_calendar.css" rel="stylesheet" />
    <script src="lht_events.js" defer></script>
    <script src="lht_calendar.js" defer></script>
</head>
```

2. Close the lht_august.html file, saving your changes.
3. Return to the **lht_calendar.js** file in your text editor, and then scroll down to the calDays() function.
4. Within the `if else` statement conditions, change the expression `+ i +` in two places to:

```
+ i + dayEvent[i] +
```

Figure 10-31 highlights the newly added code that displays the events on each day.

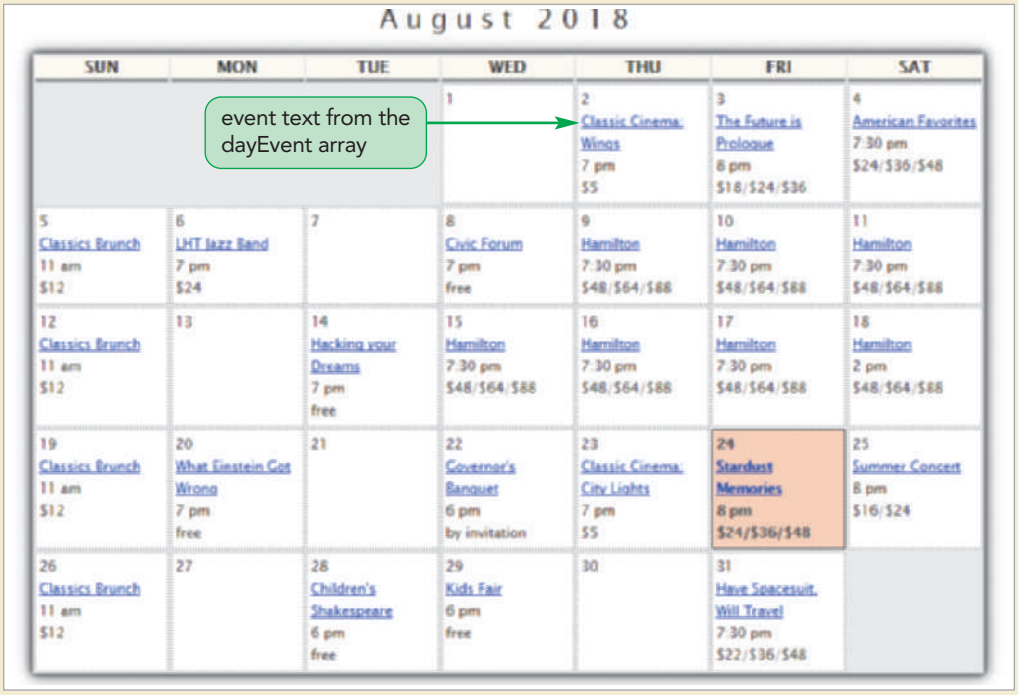
Figure 10-31 Displaying events for each day of the month

```
if (weekDay === 0) htmlCode += "<tr>";
if (i === highlightDay) {
   htmlCode += "<td class='calendar_dates' id='calendar_today'>" + i + dayEvent[i] + "</td>";
} else {
   htmlCode += "<td class='calendar_dates'>" + i + dayEvent[i] + "</td>";
}
if (weekDay === 6) htmlCode += "</tr>";
```

displays the event for the day

5. Save your changes to the file.
6. Reload the **lht_august.html** file in your browser. Verify that the calendar now shows the daily events as displayed in Figure 10-32.

Figure 10-32 Final version of the August 2018 calendar



You complete your work on the app by modifying the code so that it shows the calendar for the current month.

To display the calendar for the current month:

1. Return to the **lht_calendar.js** file in your editor.
2. Change the statement setting the value of the thisDay variable to:

```
var thisDay = new Date();
```

Figure 10-33 highlights the changed code in the file.

Figure 10-33 Displaying a calendar for the current month and date

```
/* Set the date displayed in the calendar */
var thisDay = new Date();
```

sets the thisDay variable to the current date and time

3. Close the file, saving your changes.
4. Reload the **lht_august.html** file in your browser. Verify that the page shows the calendar for the current month and that the current date is highlighted within the calendar (the events listed in the calendar will still be based on the entries in the dayEvent array).

Managing Program Loops and Conditional Statements

Although you are finished with the calendar app, you still should become familiar with some features of program loops and conditional statements for future work with these JavaScript structures. You examine three features in more detail—the `break`, `continue`, and `label` statements.

Exploring the `break` Command

Although you briefly saw how to use the `break` statement when creating a `switch` statement, the `break` statement can be used anywhere within program code. Its purpose is to terminate any program loop or conditional statement. When a `break` statement is encountered, control is passed to the statement immediately following it. It is most often used to exit a program loop before the stopping condition is met. For example, consider a loop that examines an array for the presence or absence of a particular value, such as a customer ID number. The code for the loop might look as follows:

```
for (var i = 0; i< ids.length; i++) {
   if (ids[i] === "C-14281") {
      alert("C-14281 is in the list");
   }
}
```

What would happen if the ids array had tens of thousands of entries? It would be time consuming to keep examining the array once the C-14281 ID has been encountered. To address this, the following `for` loop breaks off when it encounters the ID value, keeping the browser from needlessly examining the rest of the array:

```
for (var i = 0; i< ids.length; i++) {
   if (ids[i] === "C-14281") {
      alert("C-14281 is in the list");
      break;  // stop processing the for loop
   }
}
```

Exploring the `continue` Command

The **`continue` statement** is similar to the `break` statement except that instead of stopping the program loop altogether, the `continue` statement stops processing the commands in the current iteration of the loop and continues on to the next iteration. For example, your program might employ the following `for` loop to add the values from an array:

```
var total = 0;
for (var i = 0; i < data.length; i++) {
   total += data[i];
}
```

Each time through the loop, the value of the current entry in the data array is added to the total variable. When the `for` loop is finished, the total variable is equal to the sum of the values in the data array. However, what would happen if this were a sparse array containing several empty entries? In that case, when a browser encountered a missing or null value, that value would be added to the total variable, resulting in a null total. One way to fix this problem would be to use the `continue` statement, jumping out of the current iteration if a missing or null value were encountered. The revised code would look like the following:

```
var total = 0;
for (var i = 0; i < data.length; i++) {
   if (data[i] === null) continue;  // continue to next iteration
   total += data[i];
}
```

Exploring Statement Labels

`Statement` labels are used to identify statements in JavaScript code so that you can reference those lines elsewhere in a program. The syntax of the `statement` label is

```
label: statements
```

where *`label`* is the text of the label and *`statements`* are the statements identified by the label. You have already seen labels with the `switch` statement, but labels can also be used with other program loops and conditional statements to provide more control over how statements are processed. Labels often are used with `break` and `continue` statements in order to break off or continue a program loop. The syntax to reference a label in such cases is simply

```
break label;
```

or

```
continue label;
```

For example, the following `for` loop uses a `statement` label not only to jump out of the programming loop when the text string C-14281 is found but also to jump to the location in the script identified by the next_report label and to continue to process the statements found there:

```
for (var i = 0; i< ids.length; i++) {
   if (ids[i] === "C-14281") {
      document.write("C-14281 is in the list.");
      break next_report;
   }
}

next_report:
JavaScript statements
```

PROSKILLS

Teamwork: The Danger of Spaghetti Code

Spaghetti code is a pejorative programming term that refers to convoluted or poorly written code. One hallmark of spaghetti code is the frequent branching from one section of code to another, making it difficult to track the program line-by-line as it is executed. A change in one part of the program could lead to unpredictable changes in a completely different section of the code.

Most developers discourage the use of break, continue, and label statements unless absolutely necessary. They can confuse a programmer trying to debug code in which a program loop can end before its stopping condition, or code in which statements are not processed in the order that they are written in a document. Almost all of the tasks you perform with these statements can also be performed by carefully setting up the conditions for program loops.

Even with the best of intentions, spaghetti code can easily occur in environments in which the same code is maintained by several people or passed from one employee to another. Each programmer adds a particular feature that is needed today without adequately documenting the changes made to the code and without considering the impact of those changes on the larger program.

To avoid or at least reduce the occurrence of spaghetti code, you should always document your code and develop a structure that is easy to follow. Break up tasks into smaller functions that are easier to manage and can be reused in other parts of your programs. Also, avoid global variables whenever possible because a change in the value of a global variable can have repercussions throughout the entire code. Instead, use local variables with their scope limited to small, compact functions. If a variable must be used elsewhere in your code, it should be passed as a parameter value with the meaning and purpose of the parameter well documented within the program.

By practicing good coding techniques, you can make your programs more accessible to your colleagues and make it easier to pass your code on to your successors.

Lewis is pleased with the final version of the calendar app. Because of the way the function and the style sheets were designed, he can use this utility in many other pages on the website with only a minimal amount of recoding in the documents.

REVIEW

Session 10.3 Quick Check

1. What is a conditional statement? What is the most commonly used conditional statement?
2. Provide code to display an alert box with the message "Good Morning" if the value of the thisHour variable is less than 9.
3. Provide code to display an alert box with the message "Good Morning" if the value of the thisHour variable is less than 9 and the alert box message "Good Day" if otherwise.
4. Provide code to display an alert box with four possible messages: "Good Morning", "Good Day", "Good Afternoon", or "Good Evening" depending on whether the value of the thisHour variable is less than 9, less than 12, less than 16, or otherwise.
5. Provide the expression to extract the day of the week value from a `Date` object variable named thisDate.
6. Use a conditional operator to assign a value of "Weekend" to the thisWeek variable if thisDate equals 0 or 6 and a value of "Weekday" if otherwise.
7. What command can be used to break out of the current iteration in a `for` loop?
8. What command forces a script to go to the next iteration of the current program loop?

PRACTICE

Review Assignments

Data Files needed for the Review Assignments: lht_events_txt.html, lht_table_txt.js, 3 CSS files, 1 JS file, 2 PNG files

Lewis wants you to write another script that shows a table of events at the Lyman Hall Theater over the next two weeks from the current date. He has already created three arrays for use with the script:

- The eventDates array containing a list of dates and time at which theater events are scheduled
- The eventDescriptions array containing the description of those events
- The eventPrices array containing the admission prices of those events

Lewis has already written the page content and provided style sheets for use with the page. Your job will be to write a script that selects the events that occur in the two-week window from the current date and display them in the web page. A preview of the page you will create is shown in Figure 10-34.

Figure 10-34 Upcoming events at the Lyman Hall Theater

© Igor Borodin/Shutterstock.com

Complete the following:

1. Use your editor to open the **lht_events_txt.html** and **lht_table_txt.js** files from the html10 ▸ review folder. Enter ***your name*** and ***the date*** in the comment section of each file, and save them as **lht_events.html** and **lht_table.js** respectively.
2. Go to the **lht_events.html** file in your editor. Directly above the closing `</head>` tag, insert `script` elements that link the page to the **lht_list.js** and **lht_table.js** files in that order. Defer the loading and running of both script files until after the page has loaded.

3. Scroll down the document and, directly after the closing `</article>` tag, insert a `div` element with the ID eventList. It is within this element that you will write the HTML code for the table of upcoming theater events. Close the file saving your changes. (*Hint*: Be sure to review this file and all the support files, noting especially the names of variables that you will be using in the code you create.)
4. Go to the **lht_table.js** file in your editor. Below the comment section, declare a variable named **thisDay** containing the date August 30, 2018. You will use this date to test your script.
5. Create a variable named **tableHTML** that will contain the HTML code of the events table. Add the text of the following HTML code to the initial value of the variable:

```
<table id='eventTable'>
   <caption>Upcoming Events</caption>
   <tr><th>Date</th><th>Event</th><th>Price</th></tr>
```

6. Lewis only wants the page to list events occurring within 14 days after the current date. Declare a variable named **endDate** that contains a `Date` object that is 14 days after the date stored in the thisDay variable. (*Hint*: Use the `new Date()` object constructor and insert a time value that is equal to thisDay.getTime() + 14*24*60*60*1000.)
7. Create a `for` loop that loops through the length of the eventDates array. Use `i` as the counter variable.
8. Within the `for` loop insert the following commands in a command block:
   a. Declare a variable named **eventDate** containing a `Date` object with the date stored in the i^{th} entry in the eventDates array.
   b. Declare a variable named **eventDay** that stores the text of the eventDate date using the `toDateString()` method.
   c. Declare a variable named **eventTime** that stores the text of the eventDate time using the `toLocaleTimeString()` method.
   d. Insert an `if` statement that has a conditional expression that tests whether thisDay is ≤ eventDate and eventDate ≤ endDate. If so, the event falls within the two-week window that Lewis has requested and the script should add the following HTML code text to the value of the tableHTML variable.

```
<tr>
   <td>eventDay @ eventTime</td>
   <td>description</td>
   <td>price</td>
</tr>
```

   where *eventDay* is the value of the eventDay variable, *eventTime* is the value of the eventTime variable, *description* is the i^{th} entry in the eventDescriptions array, and *price* is the i^{th} entry in the eventPrices array.
9. After the `for` loop, add the text of the HTML code **</table>** to the value of the tableHTML variable.
10. Insert the value of the tableHTML variable into the inner HTML of the `page` element with the ID eventList.
11. Document your code in the script file using appropriate comments.
12. Save your changes to the file, and then load the **lht_events.html** file in your browser. Verify that the page shows theater events over a two-week period starting with Friday, August 31, 2018 and concluding with Wednesday, September 12, 2018.

APPLY

Case Problem 1

Data Files needed for this Case Problem: tc_cart_txt.html, tc_cart_txt.js, tc_order_txt.js, 2 CSS files, 8 PNG files

Trophy Case Sports Sarah Nordheim manages the website for Trophy Case Sports, a sports memorabilia store located in Beavercreek, Ohio. She has asked you to work on creating a script for the shopping cart page. The script should take information on the items that the customer has purchased and present it in table form, calculating the total cost of the order. A preview of the page you will create is shown in Figure 10-35.

Figure 10-35 Trophy Case Sports shopping cart

voyeg3r/openclipart; © Marie C Fields/Shutterstock; Sources: Courtesy of the Gerald R. Ford Presidential Museum; Vintagecardprices.com; Library of Congress Prints and Photographs Division; facebook.com

Sarah has already designed the page layout. Your job will be to use JavaScript to enter the order information (this task will later be handled by a script running on the website) and to write a script that generates the HTML code for the shopping cart table.

Complete the following:

1. Use your editor to open the **tc_cart_txt.html**, **tc_cart_txt.js** and **tc_order_txt.js** files from the html10 ▸ case1 folder. Enter ***your name*** and ***the date*** in the comment section of each file, and save them as **tc_cart.html**, **tc_cart.js** and **tc_order.js** respectively.
2. Go to the **tc_cart.html** file in your editor. Directly above the closing `</head>` tag, insert `script` elements to link the page to the tc_order.js and tc_cart.js files in that order. Defer the loading and running of both script files until after the page has loaded.
3. Scroll down the file and directly below the h1 heading titled "Shopping Cart" insert a `div` element with the ID cart.
4. Save your changes to the file and go to the **tc_order.js** file in your editor.
5. Within the tc_order.js file, you will create arrays containing information on a sample customer order. Create an array named **item** that will contain the ID numbers of the items purchased by the customer. Add the following four item numbers to the array: 10582, 23015, 41807, and 10041.
6. Create an array named **itemDescription** containing the following item descriptions:
   - 1975 Green Bay Packers Football (signed), Item 10582
   - Tom Landry 1955 Football Card (unsigned), Item 23015
   - 1916 Army-Navy Game, Framed Photo (signed), Item 41807
   - Protective Card Sheets, Item 10041
7. Create an array named **itemPrice** containing the following item prices: 149.93, 89.98, 334.93, and 22.67.
8. Create an array named **itemQty** containing the following quantities that the customer ordered of each item: 1, 1, 1, and 4.
9. Save your changes to the file, and then open the **tc_cart.js** file in your editor.
10. In your script, you will calculate a running total of the cost of the order. Declare a variable named **orderTotal** and set its initial value to 0.
11. Declare a variable named cartHTML that will contain the HTML code for the contents of the shopping cart, which will be displayed as a table. Set its initial value to the text string:

```
<table>
<tr>
<th>Item</th><th>Description</th><th>Price</th><th>Qty</th><th>Total</th>
</tr>
```

12. Create a `for` loop that loops through the entries in the item array. Each time through the loop, execute the commands described in Steps a through e.
    a. Add the following HTML code to the value of the cartHTML variable

```
<tr>
<td><img src='tc_item.png' alt='item' /></td>
```

    where *item* is the current value from the item array.

b. Add the following HTML code to the cartHTML variable to display the description, price, and quantity ordered of the item

```
<td>description</td>
<td>$price</td>
<td>quantity</td>
```

where *description* is the current value from the itemDescription array, *price* is the current value from the itemPrice array preceded by a $ symbol, and *quantity* is the current value from the itemQty array.

c. Declare a variable named **itemCost** equal to the *price* value multiplied by the *quantity* value for the current item.

d. Add the following HTML code to the cartHTML variable to display the cost for the item(s) ordered, completing the table row

```
<td>$cost</td></tr>
```

where *cost* is the value of the itemCost variable, preceded by a $ symbol.

e. Add the value of the itemCost variable to the orderTotal variable to keep a running total of the total cost of the customer order.

13. After the `for` loop has completed, add the following HTML code to the value of the cartHTML variable, completing the shopping cart table

```
<tr>
<td colspan='4'>Subtotal</td>
<td>$total</td>
</tr>
</table>
```

where *total* is the value of the orderTotal variable, preceded by a $ symbol.

14. Apply the cartHTML value to the inner HTML of the `div` element with the ID cart.
15. Document your script file with appropriate comments, and then save your work.
16. Open the **tc_cart.html** file in your browser and verify that the page now shows the shopping cart data for the sample customer order.

APPLY

Case Problem 2

Data Files needed for this Case Problem: hg_game_txt.html, hg_report_txt.js, 2 CSS files, 1 JS file, 4 PNG files

Harpe Gaming Sean Greer manages the development of the website for Harpe Gaming, a store chain specializing in digital games and entertainment. He is working on a redesign of the website and has asked you to work on the design of product pages. Each product page contains a description of a game and a few sample customer reviews. Figure 10-36 shows a preview of the page you will work on.

Figure 10-36 Harpe Gaming product page

Source: sixsixfive/openclipart; © Courtesy Patrick Carey

You work on a page for a digital game called *Dance Off*. The information about the game and customer reviews is stored in an external JavaScript file. Your job will be to extract that data from the JavaScript file and write it into the HTML code of the web page.

Complete the following:

1. Use your editor to open the **hg_game_txt.html** and **hg_report_txt.js** files from the html10 ▸ case2 folder. Enter ***your name*** and ***the date*** in the comment section of each file, and save them as **hg_game.html** and **hg_report.js** respectively.
2. Go to the **hg_game.html** file in your editor. Directly above the closing `</head>` tag, insert `script` elements to link the page to the hg_product.js and hg_report.js files in that order. Defer the loading and running of both script files until after the page has loaded.
3. Scroll down the document and insert an empty `article` element and an empty `aside` element directly above the closing `</section>` tag. The `article` element will contain information about the game. The `aside` element will contain a list of customer reviews.
4. Save your changes to the file, and then open the **hg_product.js** file in your editor. Take some time to review the variables and values stored in the file but do not make any changes to the file content.
5. Go to the **hg_report.js** file in your editor. First, you write information about the game that will be displayed in the web page. Declare a variable named gameReport. Within the **gameReport** variable, store the following HTML code

```
<h1>title</h1>
<h2>By: manufacturer</h2>
<img src="hg_id.png" alt="id" id="gameImg" />
<table>
  <tr><th>Product ID</th><td>id</td></tr>
  <tr><th>List Price</th><td>price</td></tr>
  <tr><th>Platform</th><td>platform</td></tr>
  <tr><th>ESRB Rating</th><td>esrb</td></tr>
  <tr><th>Condition</th><td>condition</td></tr>
  <tr><th>Release</th><td>release</td></tr>
</table>
summary
```

   where *title*, *manufacturer*, *id*, *price*, *platform*, *esrb*, *condition*, *release* and *summary* use the values from corresponding variables in the hg_product.js file.
6. Display the value of the gameReport variable in the inner HTML of the first (and only) `article` element in the document. (*Hint*: Use the `getElementsByTagName()` method, referencing the first item in the array of `article` elements.)
7. Next, you write the information from the customer ratings. Start by calculating the average customer rating of the game. Declare a variable named **ratingsSum** setting its initial value to 0.
8. Declare a variable named **ratingsCount** equal to the length of the ratings array.
9. Create a `for` loop to loop through the contents of the ratings array. Each time through the loop, add the value of current ratings value to the value of the ratingsSum variable.
10. After the `for` loop, declare the **ratingsAvg** variable, setting its value equal to the value of the ratingsSum variable divided by the value of ratingsCount.
11. Declare a variable named **ratingReport**. Set its initial value to the text string

```
<h1>Customer Reviews</h1>
<h2> average out of 5 stars (count reviews)</h2>
```

   where *average* is the value of the ratingsAvg variable and *count* is the value of ratingsCount.

12. Next, you display the content of the first three customer reviews. Create a `for` loop in which the counter goes from 0 to 2 in steps of 1. Within the `for` loop, insert the commands described in Steps a through c:
    a. Add the following HTML code to the value of the ratingReport variable

```
div class="review">
<h1>title</h1>
  <table>
  <tr><th>By</th><td>author</td></tr>
  <tr><th>Review Date</th><td>date</td></tr>
  <tr><th>Rating</th><td>
```

    where *title* is the value of the ratingTitles array item for current review, *author* is the value of the current ratingAuthors array item, and *date* is the value of the current ratingDates item.
    b. Each customer rates the game on a scale of 1 to 5 stars. Sean would like to have the stars displayed graphically. Add a nested `for` loop where the counter goes from 1 up to the value of the current customer rating of the game in increments of one. Each time through the nested `for` loop, add the following HTML code to the value of the ratingReport variable:

```
<img src="hg_star.png" />
```

    c. Directly after the nested for loop, but still within the outer `for` loop, insert commands to add the following HTML code to the value of the ratingReport variable

```
</td></tr></table>
summary
</div>
```

    where *summary* is the value from the ratingSummaries array for the current customer review.
13. Write the value of the ratingReport variable to the inner HTML of the first and only `aside` element in the document. (*Hint*: As you did with the `article` element in Step 6, use the `getElementsByTagName()` method and reference the first item from the array of `aside` elements.)
14. Document your code with informative comments throughout, and then save the file.
15. Open the **hg_game.html** file in your browser. Verify that the page shows the game summary and contents of the first three customer reviews. The page should also correctly calculate an average customer rating of 3.79 for the *Dance Off* game based on a total of 19 customer reviews.

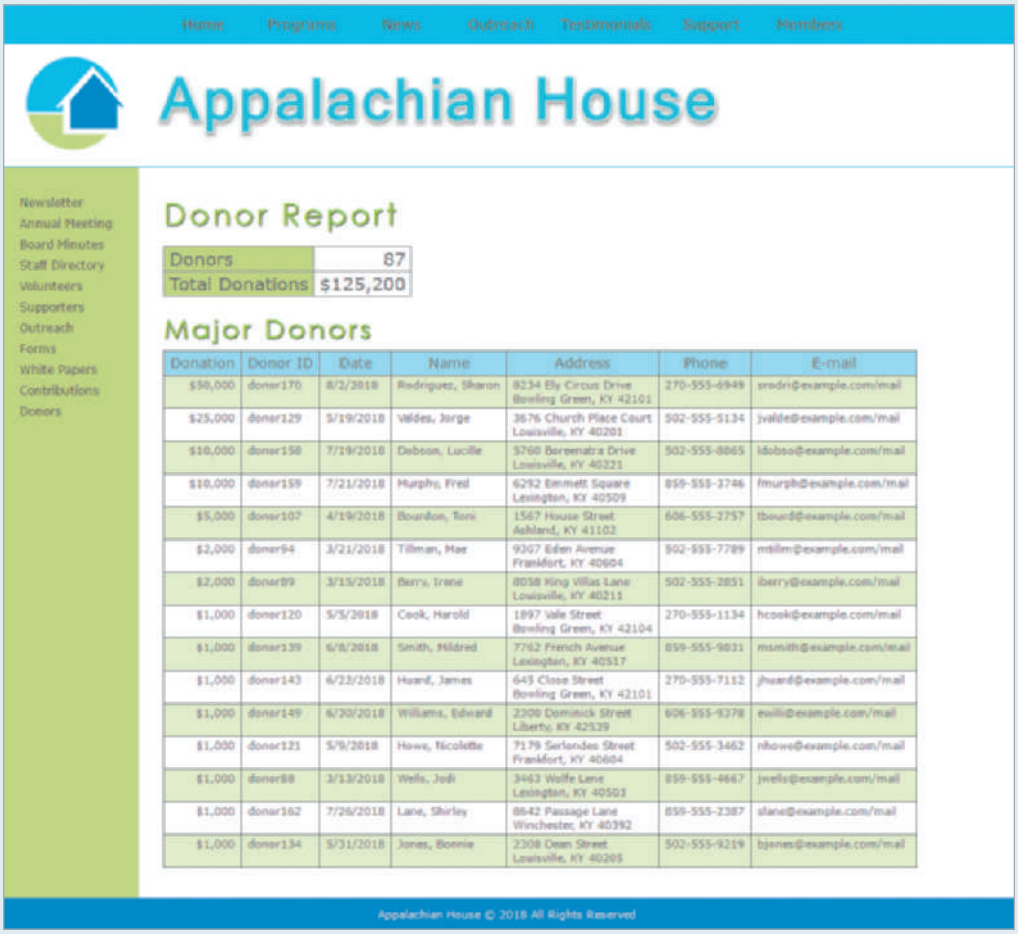
CHALLENGE

Case Problem 3

Data Files needed for this Case Problem: ah_report_txt.html, ah_report_txt.js, 2 CSS files, 1 JS file, 1 PNG file

Appalachian House Kendrick Thorne is the fundraising coordinator for Appalachian House, a charitable organization located in central Kentucky. One of his responsibilities is to report on the progress Appalachian House is making in soliciting donations. On an administration web page available only to Appalachian House staff, Kendrick wants to display a list of information on recent donations. The data on the donations has been made available to him in a multidimensional array within a JavaScript file. Kendrick wants your help in retrieving the data from this array and writing a script to produce the HTML code summarizing the result. Figure 10-37 shows a preview of the page you will create.

Figure 10-37 Donors page at Appalachian House



© Courtesy Patrick Carey

As part of writing the script for Kendrick, you work with some of the JavaScript array methods used to filter and loop through the contents of an array.

Complete the following:

1. Use your editor to open the **ah_report_txt.html** and **ah_report_txt.js** files from the html10 ▸ case3 folder. Enter ***your name*** and ***the date*** in the comment section of each file, and save them as **ah_report.html** and **ah_report.js** respectively.
2. Go to the **ah_report.html** file in your editor. Directly above the closing `</head>` tag, insert `script` elements to link the page to the ah_donors.js and ah_report.js files in that order. Defer the loading and running of both script files until after the page has loaded.
3. Scroll down the file to the h1 heading entitled "Donor Report". Directly below this h1 heading, insert the following `div` elements into which you will insert the donation summary:

```
<div id="donationSummary"></div>
<div id="donorTable"></div>
```

4. Save your changes to the file, and then open the **ah_donors.js** file in your editor. Study the content of the multidimensional array named donors. Note that the first column of the array (with an index of 0) contains the ID of each donor, the second column (index 1) contains the donor's first name, the third column (index 2) contains the donor's last name, and so forth. The amount of each donation is stored in the tenth column (index 9). Do not make any changes to the content of this file.
5. Go to the **ah_report.js** file in your editor. The file contains four callback functions at the end of the file that you will use in generating the donation report. Take some time to study the content of these functions.
6. Create a variable named **donationTotal** in which you will calculate the total amount of the donations to Appalachian House. Set its initial value to 0.

Explore 7. Apply the `forEach()` method to the donors array, using the callback function calcSum(). This statement will calculate the donation total.

8. Create a variable named **summaryTable** storing the text of the following HTML code

```
<table>
 <tr><th>Donors</th><td> donors </td></tr>
 <tr><th>Total Donations</th><td>$total</td></tr>
</table>
```

   where *donors* is the length of the donors.array, and *total* is the value of the donationTotal variable, preceded by a $. Apply the `toLocaleString()` method to the donationTotal variable so that the total amount of donations is displayed with a thousands separator in the report.
9. Set the innerHTML property of the `div` element with the ID donationSummary to the value of the summaryTable variable.

Explore 10. Kendrick wants the report to show a list of the donors who contributed $1000 or more to Appalachian House. Using the `filter()` method with the callback function findMajorDonors(), create an array named **majorDonors**.

Explore 11. Kendrick wants the major donors list sorted in descending order. Apply the `sort()` method to the majorDonors variable using the callback function donorSortDescending().

12. Create a variable named **donorTable** that will store the HTML code for the table of major donors. Set the initial value of the variable to the text of the following HTML code:

```
<table>
 <caption>Major Donors</caption>
 <tr>
  <th>Donation</th><th>Donor ID</th>
  <th>Date</th><th>Name</th><th>Address</th>
  <th>Phone</th><th>E-mail</th>
</tr>
```

13. Create the HTML code for each donor row by applying the `forEach()` method to the majorDonors variable, using writeDonorRow() as the callback function.
14. Add the text string **`</table>`** to the value of the donorTable variable.
15. Set the `innerHTML` property of the `div` element with the ID donorTable to the value of the donorTable variable.
16. Add comments to your script, documenting your work.
17. Save your changes to the file, and then open **ah_report.js** in your browser. Verify that the page shows a total donation to Appalachian House of $125,200 from 87 donors. Also, verify that the page shows a list, in descending order, of 15 donors who contributed $1000 or more to the charity.

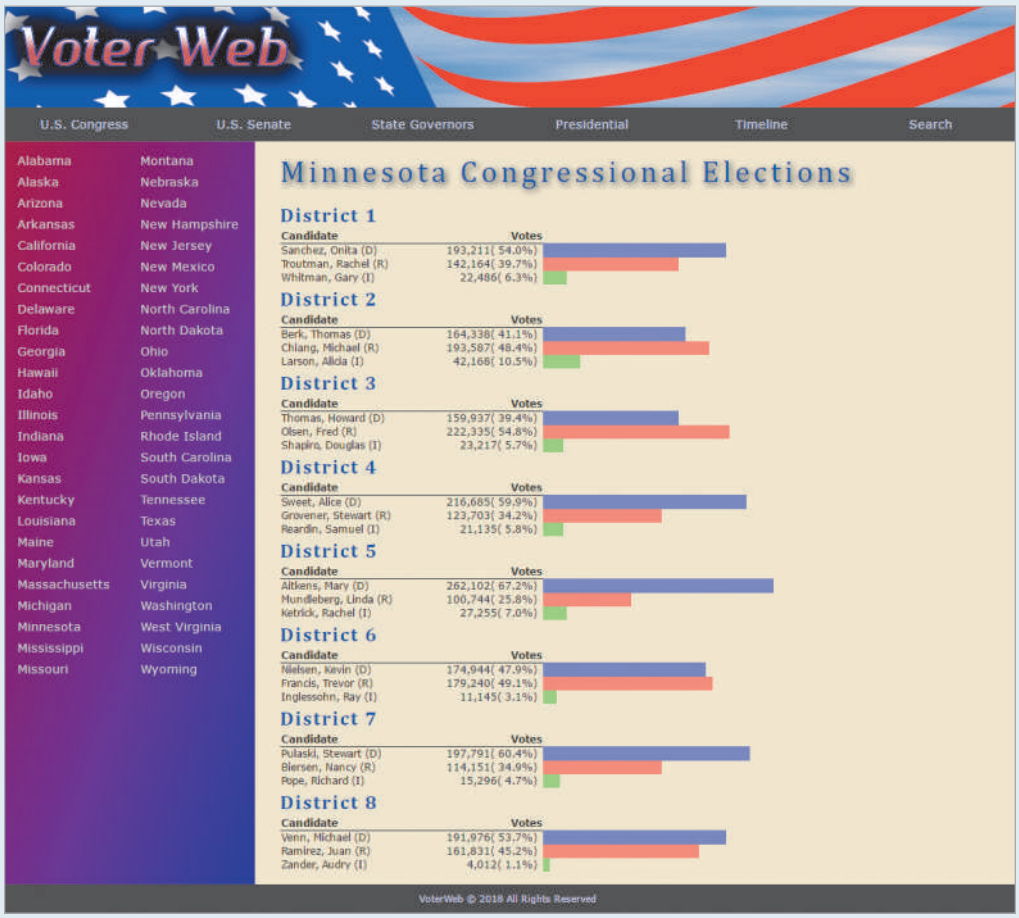
CREATE

Case Problem 4

Data Files needed for this Case Problem: vw_election_txt.html, vw_results_txt.js, 2 CSS files, 1 JS file, 1 PNG file

VoterWeb Pam Carls is a manager for the website Voter Web, which compiles voting totals and statistics from local and national elections. Pam has the results of recent congressional elections from eight districts in Minnesota stored as multidimensional arrays in a JavaScript file. Pam wants you to create a script displaying these results and calculating the vote percentage for each candidate within each race. A preview of the page is shown in Figure 10-38.

Figure 10-38 Election results at VoterWeb



Complete the following:

1. Use your editor to open the **vw_election_txt.html** and **vw_results_txt.js** files from the html10 ▸ case4 folder. Enter ***your name*** and ***the date*** in the comment section of each file, and save them as **vw_election.html** and **vw_results.js** respectively.
2. Go to the **vw_election.html** file in your editor. Directly above the closing `</head>` tag, insert `script` elements to link the page to the vw_congminn.js and vw_results.js files in that order. Defer the loading and running of both script files until after the page has loaded.
3. Scroll down the file and, directly above the footer, insert an empty `section` element. You will write the HTML code of the election report in this element. Save your changes to the file.
4. Open the **vw_congminn.js** file in your editor and study the contents. Note that the file contains the results of 8 congressional elections in Minnesota. The candidate information is stored in multidimensional arrays named candidate, party, and votes. Do not make any changes to this file.
5. Go to the **vw_results.js** file in your editor. Declare a variable named **reportHTML** containing the following HTML text

   ```
   <h1>title</h1>
   ```

   where *title* is the value of the raceTitle variable stored in the vw_congminn.js file.
6. Create a `for` loop that loops through the contents of the race array using `i` as the counter variable. Place the commands specified in Steps a through e within this program for loop:
   a. Create a variable named **totalVotes** that will store the total votes cast in each race. Set its initial value to 0.
   b. Calculate the total votes cast in the current race by applying the `forEach()` method to i^{th} index of the votes array using the calcSum() function as the callback function.
   c. Add the following HTML text to the value of the reportHTML variable to write the name of the current race in the program loop

      ```
      <table>
       <caption>race</caption>
       <tr><th>Candidate</th><th>Votes</th></tr>
      ```

      where *race* is the i^{th} index of the race array.
   d. Call the candidateRows() function (you will create this function shortly) using the counter variable `i` and the totalVotes variable as parameter values. Add the value returned by this function to the value of the reportHTML variable.
   e. Add the text **`</table>`** to the value of the reportHTML variable.
7. After the `for` loop has completed, write the value of the reportHTML variable into the innerHTML of the first (and only) `section` element in the document.

8. Next, create the **candidateRows()** function. The purpose of this function is to write individual table rows for each candidate, showing the candidate's name, party affiliation, vote total, and vote percentage. The candidateRows() function has two parameters named **raceNum** and **totalVotes**. Place the commands in Steps a through c within this function.
   a. Declare a local variable named **rowHTML** that will contain the HTML code for the table row. Set the initial value of this variable to an empty text string.
   Explore b. Create a `for` loop in which the counter variable j goes from 0 to 2 in steps of 1 unit. Within the `for` loop do the following:
      i. Declare a variable named **candidateName** that retrieves the name of the current candidate and the current race. (*Hint*: Retrieve the candidate name from the multidimensional candidate array using the reference, candidate[raceNum][j].)
      ii. Declare a variable named **candidateParty** that retrieves the party affiliation of the current candidate in the current race from the multidimensional party array.
      iii. Declare a variable named **candidateVotes** that retrieves the votes cast for the current candidate in the current race from the multidimensional votes array.
      iv. Declare a variable named **candidatePercent** equal to the value returned by the calcPercent() function, calculating the percentage of votes received by the current candidate in the loop. Use candidateVotes as the first parameter value and totalVotes as the second parameter value.
      v. Add the following HTML code to the value of the rowHTML variable

```
<tr>
 <td>name (party)</td>
 <td>votes (percent)</td>
</tr>
```

      where *name* is the value of candidateName, *party* is the value of candidateParty, *votes* is the value of candidateVotes, and *percent* is the value of candidatePercent. Apply the `toLocaleString()` method to *votes* in order to display the vote total with a thousands separator. Apply the `toFixed(1)` method to *percent* in order to display percentage values to 1 decimal place.
   c. Return the value of the rowHTML variable.

9. Save your changes to the file, and then load **vw_election.html** in your browser. Verify that the three candidate names, party affiliations, votes, and vote percentages are shown for each of the eight congressional races.
10. Pam also wants the report to display the vote percentages as bar charts with the length of the bar corresponding to the percentage value. Return to the **vw_results.js** file in your editor. At the bottom of the file, create a function named **createBar()** with one parameter named partyType. Add the commands described in Steps a through b to the function:
    a. Declare a variable named **barHTML** and set its initial value to an empty text string.
    Explore b. Create a `switch/case` statement that tests the value of the partyType parameter. If partyType equal "D" set barHTML equal to:
    ```
    <td class='dem'></td>
    ```
    If partyType equals "R" set barHTML equal to:
    ```
    <td class='rep'></td>
    ```
    Finally, if partyType equals "I" set barHTML to:
    ```
    <td class='ind'></td>
    ```
11. Return the value of barHTML.

    Next, add these empty data cells to the race results table, with one cell for every percentage point cast for the candidate.
12. Scroll up to the candidateRows() function. Directly before the line that adds the HTML code `</tr>` to the value of the rowHTML variable, insert a `for` loop with a counter variable k that goes from 0 up to a value less than candidatePercent in increments of 1 unit. Each time through the loop call the createBar() function using candidateParty and candidatePercent as the parameter values.
13. Add comments throughout the file with descriptive information about the variables and functions.
14. Save your changes to the file, and then reload **vw_election.html** in your browser. Verify that each election table shows a bar chart with different the length of bars representing each candidate's vote percentage.